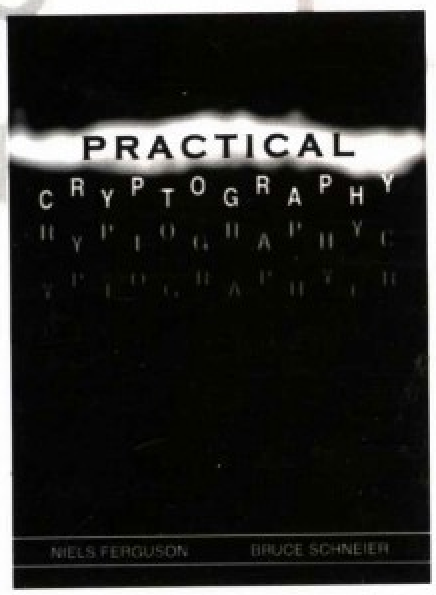
安全丛书

★ 2003年度Jolt大奖技术类入围图书

★ *Applied Cryptography* 姊妹篇

密码学实践

Practical Cryptography



[美] Niels Ferguson
Bruce Schneier 著

张振峰 徐静 李红达 等译

電子工業出版社
Publishing House of Electronics Industry
www.phei.com.cn

PDG

密码学实践

Practical Cryptography

两位世界顶尖的密码专家教你如何保护未来数字系统的安全

今天，安全问题是商业领域关心的首要问题。没有安全的计算机系统，就不能盈利，不能拓展，甚至不能从困境中走出。作为提供信息空间安全性的技术，密码学大有前途。然而令人吃惊的是，在本书出版之前还没有一部关于如何实现密码系统以及如何把它整合在实际系统中的著作。

本书是两位国际知名的作者所著的第一部实用的密码产品实现指南，填补了密码学理论与实际密码应用之间的鸿沟。

本书是最畅销的《应用密码学》（“Applied Cryptography”）的后续指南，旨在讲解密码学的基本知识。

本书主要阐述了下列问题：

- 从分组密码到数字签名的选择和使用密码模块的实用准则
- 在当今的计算机中以安全的方式来实现密码算法和密码系统
- 采用一致的设计原理以确保系统的各个部分都达到要求的安全等级
- 为什么说安全性会影响系统的所有部分，以及为什么必须把安全性看做是首要的设计目标
- 简单的密码模块接口如何能够降低系统的复杂性及提高系统的安全性

Niels Ferguson：密码工程师和顾问。他在设计和实现密码算法、协议以及大规模安全设施方面有着非常丰富的经验。Ferguson以前是DigiCash和CWI的密码学专家，并曾在Counterpane Internet Security公司与Bruce Schneier一起工作，曾发表过许多学术论文。

Bruce Schneier：Counterpane Internet Security公司的创始人和首席技术官。世界著名的科学家、安全专家和演讲家，“Secrets and Lies: Digital Security in a Networked World and Applied Cryptography”和“Applied Cryptography”的作者。

ISBN 7-121-01628-1

9 787121 016288 >

传播教育信息 共享教育资源
华信教育资源网
www.hxedu.com.cn
欢迎登录 获取优质教学资源

WILEY

PHEI

责任编辑：周宏敏
特约编辑：王　崧
责任美编：毛惠庚

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书

ISBN 7-121-01628-1　定价：33.00 元

第1章　我们的设计原理

本书是一本关于安全的书——是一本如何构建安全密码系统的书。本书体现出我们对于密码学的狂热，这是因为在这一领域工作的这些年来，我们已经看到了安全的整体状况。的确是这样的，我们分析过的每一个系统都以某种方式被攻破了。其中总是有一些部件是好的，但是它们总是以不安全的方式被使用。

假如整个社会希望未来的数字世界是安全的，那么就需要进行改进。我们期待本书能够对此有所贡献。

本书给出了大量的关于密码系统的实践信息，但是这些都不重要，除非我们能够说服你：正确地部署安全性非常重要。正确部署在其他领域意味着不留情面。这是很难做到的，我们需要花费多年的时间才能做到足够的不留情面。只有一点点安全性是不够的，这就像在院子周围修建一半的围墙，或者只把前门锁起来而把后门敞开着。安全性是不能妥协的系统性质。围墙上有一个洞就意味着不安全。所以，我们要放弃其他任何事情，给安全性留下足够的空间。根据经验，我们知道这在IT行业很难被接受。然而，如果希望使我们的数字世界安全，就必须这样做。

1.1　追求性能带来的恶果

苏格兰的Forth海湾大桥作为19世纪的工程奇迹，同从上面穿过的列车相比，它超乎想像地庞大（因而昂贵）。它是一个不可思议的超级工程，使我们很难相信自己的眼睛。而它的设计者做了一件正确的事情。他们面临的是一个之前从没有成功解决的问题：修建一座庞大的钢铁大桥。他们做出了令人震惊的工作。他们的成功是令人注目的：在100多年后的今天，这座大桥仍然在使用。一个成功的工程看起来就是这样的。

多年来，桥梁设计师已经知道了如何以更低的代价和更高的效率来修建类似的大桥。但是，首先需要考虑的是要建造一座安全而且实用的大桥，以缩减成本的方式带来的效率则是次要的问题。

在计算机安全中我们则颠倒了这些先后顺序。首要的设计目标时常是严格的效率需求，第一优先考虑的事情始终是速度，即使在速度并不重要的地方也是如此，这样就削减了安全方面的花费。而在安全性这个工程领域中，我们没有构建安全系统的技巧，即使拥有无限的预算。其结果总是构建一个高效的系统，但也不可避免地是一个不安全的系统。

我们从另一个角度来看一看苏格兰Forth海湾大桥的故事。在1878年，Thomas Bouch建成了当时世界上最长的大桥，它横跨敦提的泰河海湾。Bouch采用了铸铁和熟铁，使用了一种新的设计方法，这座大桥被认为是一个工程奇迹。建成不到两年，在1879年12月28日的夜里，大桥在一场暴风雪中倒塌了，当时正好有一列承载着75个人的列车经过，列车上所有的人都遇难了。这在当时是一场巨大的工程灾难①，所以，几年后当设计Forth海湾大桥的时候，设计者使用了更多的钢铁，这样做不仅使得大桥更安全，而且使公众看起来也觉得安全。

我们知道，有时候工程师的设计是错误的，尤其是当他们进行一些新的尝试的时候。而当他们设计出错的时候就会有人为此而丧生。然而，维多利亚的工程师们有一条很好的经验：如果失败了，就退回去并且采用更加保守的方法。计算机行业忘记了这个经验。在计算机系统中，当遭遇了非常严重的安全失败的时候（几乎每周都有这类事情发生），我们只是迈着沉重的脚步向前走，并把它看做是注定要发生的事情来接受。我们不会回到制图板并更加保守地进行设计，只是持续不断地抛出一些补丁并期望它们会解决问题，这样做是很不光彩的。

现在已经非常清楚了，任何时候我们都要选择安全性而不是效率。我们愿意在安全性上花费多少CPU的时间呢？——几乎所有的时间。如果在可靠的安全系统上花费90%的CPU时间，我们也不会在意。对于我们以及大多数用户来说，缺乏安全性是真正的伤害。这就是人们为何仍然通过传送几页纸来进行签名，为何必须担心病毒以及针对计算机的其他攻击的原因。未来的数字骗子知道的事情会更多，装备也会更好，从而计算机安全将成为一个越来越大的问题。我们所看到的只是数字犯罪潮的开始。如果想使用因特网来进行商业交易，就必须更好地保护我们的计算机。

当然，有很多方式可以实现安全性。但是，正如Bruce在《秘密与谎言》一书中用大量资料详细描述的那样，有效的安全性总要与防护、检测和响应相结合[89]②。密码系统的作用属于防护部分，它必须非常好，从而保证检测和响应部分（它们能够并且应该相互干涉）不会被控制。不管怎样，本书是关于密码学的，所以我们将把注意力集中在这上面。

是的，我们知道，你还对前面提到的90%感到吃惊。但是，我们的计算机还应该做什么事情呢？我们每秒只能打出大约10个字符——指的是现在——而即使在一个世纪之前的低速机器上做到这一点也没有什么问题。如今的机器要快上1000多倍。如果我们把90%的CPU用于安全性，计算机的速度看起来只剩下十分之一了。这大约是五年以前的计算机速度，而那些计算机对我们完成工作来说已经足够快了。

只有少数情况我们必须坐在计算机前面等待，包括等待打开网页、打印数据、启动某些程序、启动机器等。好的安全系统不应该使这些活动慢下来。现代的计算速度如此之快，很难估计如何以有益的方式来利用这种高速度。当然，我们可以在图像处理、三维动画乃至语音识别中使用，但是这些应用的关键部分不会执行任何与安全有关的动作，所以它们不会由于引入安

① William McGonagall为此写了一首有名的诗，结尾这样写道：建造的房子越坚固，受到伤害的可能性就越小。这样的建议在今天也是非常中肯的。

② 表示参见参考文献[n]。这里，n为89。

全系统而慢下来。正是其余的一些系统会占用开销，但它们已经尽可能快了。如果它们减慢了10倍，我们并不在意。大多数时间你甚至没有注意到这些开销。即使对于开销很重要的情况，换取的也只是商业代价。

如果你曾经想以效率的名义切下安全的一角，就需要对自己不断地说："我们已经有了足够快速但却不安全的系统，我们不需要另外一个"。我们发现人们乐意接受这种解释，而不是长篇大论的关于安全等级的理由。

现在已经很清楚了，我们考虑的优先顺序是：安全性第一，安全性第二，安全性第三，而性能则在列表下面的某个地方。当然，我们还是希望系统的效率尽可能地高，但它不是以安全性为代价换来的。我们知道这一设计原理在现实世界里并不总是可行的，通常市场需求会胜过对安全的需求。很少有系统是从零开始开发的，常常需要逐渐地增加保护，或者是在部署之后加以保护。系统需要与现有的不安全系统反向兼容。我们两个人都在这些限制条件下设计了很多安全系统。可以告诉读者的是，按照那种方式几乎不可能构建一个好的安全系统。本书的设计原理是安全性第一，安全性至上。我们愿意看到这一原理在商业系统中被更多地采用。

1.2 追求性能带来的恶果

没有一个复杂的系统是安全的。复杂性是安全性的最大敌人，而复杂性总以特性或者选项的形式出现。

我们将在以后更详细地讨论为什么说复杂性不能与安全性结合在一起，不过这里先给出一个基本的理由。设想一个有着20个不同选项的计算机程序，每一个选项可以选中也可以不选，那么就会有100多万个不同的组合。为了使程序能够工作，只需要对最常用的选项组合进行测试。而要保证程序是安全的，就必须对程序可能具有的100多万个组合中的每一个进行评估，并检查各个组合在每一种可能的攻击下是安全的，这是不可能做到的。而多数程序都不止20个选项。如果想要构建安全的东西，就必须把它简化。

简单的系统不一定是小系统。我们既可以构建大系统，同时也可以使其相当简单。复杂性用来度量在任何一个点有多少事件相互作用。如果一个选项的影响只限于程序的一小部分，那么它就不可能与一个影响范围局限于程序另一部分的选项相互影响。为了构建大的、简单的系统，必须在系统的不同部件之间提供一个很清晰而又很简单的接口。程序设计师把它称为模块化。一个好的简单接口可以把系统的其余部分与模块的细节隔离开来。

我们其实不必写下所有这些内容，它们都是基本的软件工程方面的知识。遗憾的是，在现实世界的系统中我们很少看到这样的系统。

在本书中，我们尽力要做到的一件事情是为密码模块定义简单的接口。没有特性、没有选项、没有特殊情况，也没有其他的东西需要记忆，只有我们提出来的最简单的定义。其中有一些定义是全新的，是在写这本书的时候提出来的。它们有助于对我们的安全系统进行思考，并且希望对你也有所帮助。

第2章　密码学背景

密码学是加密的艺术和科学。至少，它的出发点是这样。如今，它的内容非常宽广，涵盖了认证、数字签名以及很多更基本的安全功能。它仍然是一门艺术和科学：要创建一个好的密码系统，要求具有科学的背景和正确地使用我们称之为经验的法术。

密码学是一个非常宽广的领域。在密码学会议上能够触及的话题范围非常广，包括计算机安全、高等代数、经济、量子物理、民法和刑法、统计、芯片设计、软件最优化、策略、用户接口问题，以及其间的任何事情。本书只关注密码学中很小的一部分，即实践方面。我们希望能够说明如何在真实世界的系统中实现一个密码系统。

密码学的宽广性和多样性使其成为一个非常迷人的工作领域，它是各种不同领域的混合体。总是有一些新东西要去学习，并且有来自于各个方面的新思想。这也是为何密码学如此困难的原因之一。一个人不可能理解所有的东西，世界上没有人知道关于密码学的所有事情，甚至没有人知道其中的大多数内容。我们当然不知道关于本书主题的所有内容，所以这就是你关于密码学的第一次课程：保持一个批判的头脑。不要盲目相信一切，即使是出版物。

2.1　密码系统的作用

密码系统本身是没有什么价值的，它必须成为一个大系统的一个部件。我们喜欢把密码系统与物理世界中的锁进行比较。锁本身是一个很无用的东西，它需要作为一个更大的系统的一个部件。这个大系统可以是建筑物的门、一个链条、一个保险箱，或者是其他的东西。锁只是更大的安全系统的一个小部件。密码系统也是一样的，也只是更大的安全系统的一个小部件。

即便密码系统只是安全系统的一个小部件，但它却是一个极其关键的部件。密码系统是一个能够让一些人访问系统而其他人不能访问的部件——这是很微妙的。安全系统的绝大部分，比如其中的围墙和篱笆，其设计目的是为了把所有人都挡在外面。密码系统扮演了锁的角色：它必须区分“好”的访问和“坏”的访问。这比把所有人都拦在外面要困难得多。因此，密码系统及其相关组件自然成为了任何安全系统的攻击点。

这并不意味着密码系统总是一个系统的脆弱点。这样的情况很少出现。在实际生活中，即使糟糕的密码系统也总是比安全系统的其他部件好一些。你可能见过银行金库的门，至少在电影里见过。要知道，它是一个10英寸厚的硬化钢，使用巨大的金属插销上锁。我们经常看到，在数字世界里，一个作用相同的金库大门被安装在帐篷上，站在旁边的人们正在讨论这个门该

有多厚，而不是花点儿时间来看看这个帐篷。人们喜欢争论密码系统的确切密钥长度，但是对修补 Web 服务器的缓存溢出却很少有兴趣。结果是可以预见的：攻击者发现了一个缓存溢出，而不用费心去攻击密码系统。只有当系统的其余部分也安全的时候，密码系统才是真正有用的。

然而，有一个原因说明了即使系统有其他的弱点，设计正确的密码系统仍然很重要。一个攻击者攻破密码系统被发现的概率很低，不会留下攻击的痕迹，攻击者的攻击看起来确实像一个"好"的访问。这就好比现实生活中的非法进入。如果夜贼使用橇棍闯进来，那么至少能看到有非法入侵发生了。但是，如果夜贼选择了锁，你可能绝想不到发生了入室行窃。很多方式的攻击会留下痕迹，或者以某种方式扰乱了系统。对于密码系统的攻击是短暂而不可见的，这使得攻击者能够一次又一次地返回。

2.2　最脆弱的链接性质

用很大的字体打印下面的句子，并把它贴在你的显示器上。

一个安全系统只能与其最薄弱的链接一样坚固。

每天看一看这句话，设法理解这句话意味着什么。最薄弱的链接是为何得到一个有效的安全系统如此困难的主要原因。

每个安全系统都是由大量不同的组件构成的。我们必须假定对手非常聪明，他将从最薄弱的地方来攻击系统。其他地方有多坚固都无关紧要。就像一个链条，最薄弱的链环将先被打破。链条中的其他链环再坚固也没有用。

Niels过去常常在一栋办公大楼里工作，所有办公室的门每晚都上锁。听起来非常安全，是吗？惟一的问题在于，这栋建筑有一个假的天花板。人可以爬到天花板上，从而爬过任何门或者墙。如果取走这些天花板，下面看起来就像一组上面有门的高高的小屋，这些门都上了锁。当然，把门上锁会给窃贼增加一点儿困难，但同时也使得安全警卫们在值夜班时检查这些房间变得更加困难。锁上门之后，整体的安全性是提高了还是变得更糟糕了，这一点是不清楚的。在这个例子中，最薄弱的链接性质使得给门上锁变得无效。它可能加强了某一个特殊的链接（如门），但是有一些其他的链接（如天花板）仍然是薄弱的。把门上锁的整体效果是很小的，而它的负面影响可能超过了其正面的作用。

为了提高系统的安全性，我们必须改善最薄弱的环节。但要做到这一点，我们必须知道这些链接都是什么，知道哪一个链接是薄弱的。最好使用分层的树状结构来进行分析。每一个系统都有多个链接，而每一个链接也都有子链接。可以把这些链接按照我们称之为攻击树的方式组织起来。图 2.1 给出了一个例子。比如说，我们想要侵入一家银行的金库。第一个链接是墙、地板、门及天花板，攻破其中任何一个都能进入金库。现在我们更详细地查看它的大门。大门系统有自己的链接：门框和墙之间的链接、锁、门本身、把门钉入门框的螺钉以及合叶。我们

接下来讨论如何攻击门锁，其中的一种办法是拿到钥匙，而这将产生一个完整的树，它表明以某种方式来窃取钥匙。

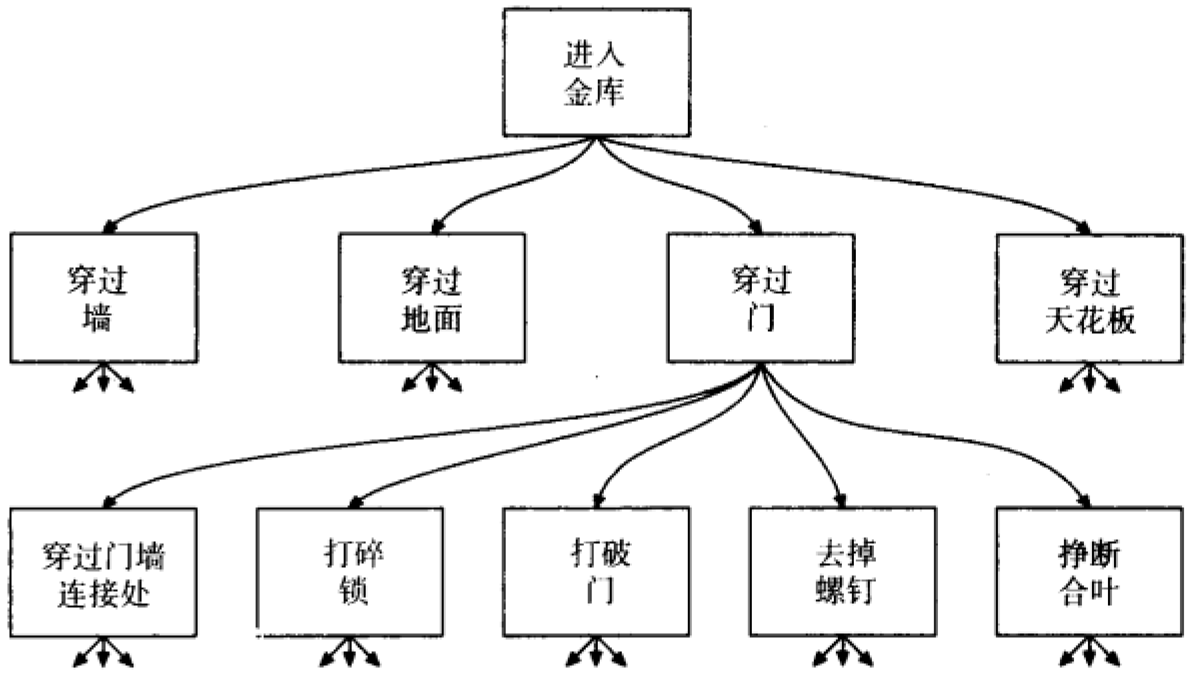


图 2.1 金库的攻击树

我们可以对每一个链接进行分解，把它分解成其他的链接，直至分解为单一的组件。对一个实际系统来说，要做到这一点可能需要大量的工作，但是对于可能的攻击路线来说，它能够提供很有价值的分析。如果在没有进行这些分析的情况下试图保护一个系统，则经常会导致无用的工作。本书只讨论有限的组成部分——那些可以使用密码学解决的部分——由此产生的攻击树非常简单，这里就不再详细描述了。

最薄弱的链接性质在很多方面影响了我们的工作。比如，我们总是假定用户拥有正确的口令，但实际上他们没有。他们会选择一些简单的短口令。用户几乎会使用任意长度的口令，从而不让安全系统给自己带来麻烦。把口令写在小贴片上并把它贴在显示器上，就是他们可能会做的诸多事情之一。绝不能忽视诸如此类的问题，因为它们总会影响最终的结果。如果你设计了一个系统，每周给用户一个新的12位数的随机口令，那么相信他一定会把它贴在显示器上面。这进一步削弱了已经非常薄弱的链接，对于整体的安全性而言是很有害的。

严格地讲，除了最薄弱的链接之外，强化任何东西都是无用的。在实际中，情况并非如此清晰。对于不同类型的攻击者而言，最薄弱的链接是不一样的。任一链接的强度都依赖于攻击者的技能和工具。所以，哪一个链接是最薄弱的要视情况而定。因此，加强每一个在特定情形下可能是最薄弱的链接都是值得的。

2.3 敌手设置

安全系统和几乎所有其他类型的工程之间的最大区别在于敌手的设置。大多数工程师必须对付诸如风暴、受热、磨损和破裂之类的难题。所有这些因素都影响了设计，但是这些影响对

于一个富有经验的工程师来说都是可以清楚地预知的。在安全系统中就不是这样了。我们的对手是智能的、机灵的、恶意的和狡猾的，他们可能会做出一些以前没有人想到过的事情，他们不按照规则行事，是完全不可预知的。这是一个更加难以对付的环境。

我们很多人还记得，在一部电影中，Tacoma Narrows 浮桥在一场狂风中摇晃扭曲，直至倒塌而跌入水中。这是一段著名的电影片段，这次倒塌给了桥梁工程师们一个很有价值的教训。脆弱的浮桥在狂风中会以共振的方式使得整个结构发生振荡，最终毁坏。在构架新桥时他们如何避免同样的事情再次发生呢？有效地加固桥身使之能够抗击振荡的代价是很高的。最常用的技术是改变桥的空气动力学。把桥面建造得更厚一些，使得风不能很容易地就把桥面上下吹动起来。有时使用栏杆作为扰流器，使得桥面不再像一只翅膀一样在风中起伏。这样做是有效的，因为风是可以明确地预知的，它不会主动地改变其行为方式来毁坏大桥。

安全工程师必须要考虑怀有恶意的狂风。如果狂风上下吹动而不是只从一面吹来该怎么办呢？如果它改变方向以产生正确的频率使得大桥发生共振又该怎么办呢？桥梁工程师不会理会这一类疯狂的议论："别傻了，风不会那样吹的"。这当然使得桥梁工程师的工作更加容易了。密码学家就没有这样的奢望了。安全系统要面对机智而怀有恶意的攻击者的攻击。我们必须针对所有类型的攻击提供保护。

敌手的设置是要引进一个非常苛刻的环境。在这个游戏中没有规则，并且对我们来说桥面是容易弯曲的。我们在抽象的意义下来谈论一个"攻击者"，但是不知道他是谁，他知道什么，他的目的是什么，他何时发起攻击，也不知道他所使用的方法是什么。由于攻击会在我们设计系统很久以后发生，他可以有五年或者十多年的研究优势，并且可以使用我们所不知道的未来的技术。而且，利用所有这些优势，他只需要找到系统中的一个脆弱点，而我们却要保护所有的区域。尽管如此，我们的任务是构建一个能够抵抗所有攻击的系统。欢迎进入密码学的神奇世界！

2.4　实用的妄想狂

要在这个领域内工作，你必须变得狡猾。你必须像一个恶意的攻击者那样进行思考，以发现自己工作的弱点。这也将影响到你的一生。每一个工作于实际密码系统的人都经历了这个过程。一旦你开始考虑如何攻击系统，你会使用周围的任何东西。你突然发现你是如何欺骗周围的人们，而他们又是如何欺骗你的。密码学家是专业的妄想狂。不久之后，你就学会了把专业的妄想与现实世界的生活分开，否则你就会疯狂了。我想……我们大多数人都会设法使自己保持头脑清醒①。

妄想狂在这个工作中是非常有用的。假设你参与研发一个电子支付系统。在这个系统中涉及到几个参与方：客户、商家、客户的银行和商家的银行。因为很难判断威胁是什么，所以我

① 但是请记住：你不是妄想狂并不意味着别人也这么想。

们使用妄想狂的模型。对于每一个参与方，我们假定其他任何人都是要欺骗这个参与方的一个大阴谋团伙的成员。如果你的密码系统在这样的妄想狂模型中可以幸存，那么它在现实世界中至少有生存下来的机会。

2.4.1 攻击

专业的妄想是这个行业的基本手段。对于所面临的任何新系统，你所想到的第一件事情就是如何攻破它。越快找到其弱点，你对这个新系统就了解得越快。没有什么比这样的事情更糟糕了：在使用一个系统多年之后，有一个人走上前来说“但是，如果我以这种方式进行攻击该怎么办……？”我们实在不想经历那个措手不及的时刻。

在这个领域内，我们对攻击某个人的工作和对某人的人身攻击进行了严格的区分。任何工作都是一个公平的游戏。如果某人提出了某个东西，这就是自动地邀请其他人来攻击它。如果你攻破了我们的一个系统，我们会称赞你的攻击并告知所有人①。我们坚持不懈地寻找系统的弱点，因为这是获知如何构建更安全系统的惟一方法。必须了解这一点：对于你的工作的攻击并不是对于你本人的攻击。如果不知道这一点就不能在这个领域生存下来，所以，习惯它吧。同样，当攻击一个系统时，要始终确信你是在批评该系统，而不是它的设计者。同任何其他领域一样，在密码学中的人身攻击将会带给你同样的负面响应。

密码学家们对于敌手的态度让那些对这一领域不甚熟悉的人们造成了误解。外面的人们有时认为，当我们攻击某个人的工作时就是在攻击他本人，从而抱怨我们没有礼貌。比如，在1999年的一次会议期间，人们展示了AES分组密码的提议。分组密码Magenta的提案人不能亲自参加，就请一个学生来代表他。结果，Magenta存在严重的弱点，在初次陈述完毕之后的五分钟提问时间内，该提议被听众中的几个人攻破。非常有趣。一些国际上最好的密码学家也出席了这个新的分组密码的讨论，他们也参与讨论并寻找弱点，而其他的听众则在旁观。我们就是这样工作的。听众中的另一个人在密码学领域中的经历较少，他撰写了一篇关于参加会议的文章。他认为那个学生受到了人身攻击，并且对于听众不可思议的无礼行为深感不安。他的反应是可以理解的。在许多事情上，对一个思想或者提议的攻击意味着对于其作者的攻击。但在密码学界中，我们不能持有那样的态度。

比这类事件更糟糕的情况是这一领域之外的人误入密码学领域。他们把对其工作的批评看做是人身攻击，并由此带来一些问题。有时需要讲究说话的策略，但是那样会使别人很难明白你的意思。说一些类似“在安全方面可能存在一些问题”之类的含糊的话，得到的回答是“哦，我们会修补它的”，然而其基本的设计很可能是不可行的。经验表明，明确问题的惟一方式就是详细而精确地讲出来，说一些诸如“如果这样做那么系统就被攻破了”之类的话。如果是与你谈话的人设计了系统，那就有问题了。然而，这个问题没有巧妙的解决办法。

① 根据你的攻击，我们可能抱怨自己没有发现该弱点，但那是另外一个问题。

所以，如果下一次有人攻击了你的工作，不要把它当成是人身攻击。而当你攻击一个系统时，要明确你是在批评系统本身，而不是它的设计者。

2.5　威胁模型

每一个系统都可能受到攻击。就安全系统整体来说，就是规定某些人可以访问系统，而某些人则不能访问。最终，总是要以某种方式体现出信任某些人，而这些人反过来也可以攻击系统。

有一点非常重要：知道你要尽力保护的是什么。这听起来好像是一个简单的问题，但实际上它比你想像的要困难得多。什么是威胁？大多数公司使用防火墙来保护他们的LAN，但是多数真正有害的攻击都是由内部人员发起的，而防火墙一点儿也不能防止内部人员的攻击。无论你的防火墙有多好，都不能防止一个怀有恶意的雇员。这是威胁模型中的错配。

另一个例子是SET。SET是用于使用信用卡进行在线购物的协议。它的一个特性是加密信用卡号使得窃听者不能复制它。这是一个好主意。第二个特征是即使商家也不能看到客户的信用卡号，但很少被使用。

第二个特征以失败告终，原因在于某些商家使用信用卡号来查询客户的记录或者索取额外的费用。整个商务系统都建立在商家可以访问客户信用卡号的假设的基础上，因此SET就试图取消这种访问权限。当Niels从事SET方面的工作时，有一个选项是两次发送信用卡号——一次是把加密的信用卡号发送给银行，另一次是对它进行加密然后发送给商家从而使商家也能得到它。（SET的规范非常庞大而复杂，我们没有下功夫查看它现在是否还是这样。）

然而，即使有这个选项，SET解决的也是一个错误的问题。大多数被盗取的信用卡号不是在客户与商家之间传送时被拦截的，它们是从商家的数据库里被窃取的。SET只是在传输过程中保护信息。

SET还犯了另一个更为严重的错误。最近，Niels在荷兰的银行提供基于SET的信用卡服务。为在线购物增加安全性是一个主要卖点，但这实际上是一个虚假的理由。使用普通的信用卡在线定购是相当安全的。你的信用卡号不是秘密，在购买东西时你把它给了每一个卖方。真正的秘密是你的签名，它才能批准交易。如果商家泄露了信用卡号，你可能会拿到奇怪的账单，但是，只要没有手写的签名（或者PIN号），就意味着不必接受该交易，因此该账单也就没有法律依据。在多数裁决中只需要进行控诉并拿回你的钱即可。使用不同的号码办理一个新的信用卡可能会带来一些麻烦，但这是用户的意愿。使用SET情形就不一样了。在SET中，用户使用数字签名（将在第13章中解释）来授权交易，这显然比只使用信用卡号更安全。但是考虑一下，现在用户要对其PC上的SET软件执行的每一次交易负责任，这使用户承担了很大的责任。如果病毒入侵了PC并暗中破坏了SET软件怎么办？软件可能对一次错误的交易进行签名，从而致使用户丢失钱财。

所以从用户的观点来看，同普通的信用卡相比，SET的安全性更差。对于在线交易来说，普通的信用卡是安全的，因为用户总是能够从一桩诈骗性的交易中拿回自己的钱。使用SET

增加了用户的风险。因此，尽管整个支付系统更安全了，但SET把剩下的风险由商家或者银行转给了用户。它把用户的风险模型由“只有当他们能够很好地伪造我的签名时才能取走我的钱”改变为“只有当他们能够很好地伪造我的签名或者精巧的病毒感染了我的PC时才能取走我的钱”。

威胁模型是重要的。当开始研发一个密码安全项目时，要坐下来考虑所要保护的风险是什么。错误的风险分析会致使整个项目毫无意义。本书我们不过多地谈论风险分析，因为我们只探讨密码学这一有限的领域。但是，在任何实际系统中，绝不能忘记对每一个参与方的风险分析。

2.6 密码系统不是解决方案

密码系统不是安全问题的解决方案。它是解决方案的一部分，或者是问题的一部分。在许多情况下，密码系统使得问题变得更严重，而且甚至根本不清楚采用密码系统是否是一种改进。

我们如何来保护汽车免于被盗呢？最容易的方法是直接用物理的方式保护汽车，即不让任何人靠近它就可以了。这有些困难，于是改为在车上安装一把锁，然后把车钥匙带在身上。由于汽车上了锁，我们就可以把它停放在大街上。窃贼有两种方式可以得到汽车。他可以撬开或者破坏车锁系统（包括敲碎车窗），也可以窃取车钥匙，而车钥匙是由你直接保护的。我们注意到，现在有两条攻击路线：车锁系统和钥匙。引入钥匙给攻击者带来了一条新的攻击路线。车钥匙的主要优点在于，保护一把车钥匙比保护一辆汽车要容易得多。只要这个优势超过了窃贼攻击锁的风险，安全性就因为有了车钥匙而得到提高。

在数字世界里，情况稍有不同。假设计算机里有一份秘密文件，而你不希望其他人读取它。你可以通过授权访问来保护文件系统，也可以加密文件而保护密钥。加了密的文件就不再是保密的，正如人类社会一样，你或许不需要很好地保护它，就像把汽车停放在大街上一样。但是，哪里可以存储密钥呢？一个好的密钥很长且很难记忆。一些应用程序把密钥存储在磁盘上——正好是存储秘密文件的第一个地方。现在攻击者有两种方式可以获取文件。能够在第一种情形中获得秘密文件的任何攻击可以马上得到密钥，进而用它来解密文件。在第一种情形中适用的任何攻击在第二种情形中也适用，但是我们引进了一个新的攻击点：攻击者可以自己攻破加密系统。最终，整体安全性降低了。因此，加密文件不是完整的解决方案。它只是部分解决方案，而且它本身带来的问题比它解决的问题更多。

密码系统有很多用途。它是一个好的安全系统的关键部分。如果使用的方式不正确，它可以使系统更脆弱。很多情况下，它提供的只是一种安全感，而不是实际的安全性。这是我们经常需要的，因为大多数客户想要这样的感觉。他们希望感到安全，但是不愿意就实际的安全性进行争论。在类似的情形（这些情形普遍存在）中，密码系统本身是一个解决方法，而客户相信的任何魔法也都能提供同样的安全感并同样能够工作。

2.7　棘手的密码系统

密码系统的实现是极其困难的，即使经验丰富的专家也会设计出一些几年后被攻破的系统。这种现象太常见了，当它发生的时候我们都不会感到惊奇。最脆弱的链接性质和敌手的设置等因素结合在一起使得密码学家的工作非常困难。

另一个重要问题是缺乏测试。没有已知的方法可以测试一个系统是否安全，我们能够做的最好的办法是公布该系统然后等待其他专家来分析它。我们注意到后者不是自动的，有很多已公布的系统自其发布以来甚至没有人看它一眼。即便会议文章和期刊文章也只是受到最基本的评审。这些评审过程旨在筛选出好的文章和一般的文章，而不是检查其结果的安全性或正确性。有一些检查是要做的，但是一般的评审人没有时间对一篇文章做全面的分析（毕竟，评审人很少因为这些工作得到报酬，而当他们利用业余时间对 20 ~ 30 篇文章进行评审时，他们不会进行非常全面的检查）。

我们对于密码学中的某些小的领域有很好的理解。这并不是说它们简单，而是说明我们对这方面的工作已经历了几十年，因此认为已经了解这些问题了。本书的内容主要是关于这些领域的。我们在本书中尽力要做的是收集关于实际密码系统的设计和构建方面的信息，并把它们放在一起。

出于某些原因，知道一点儿密码学的人认为它是容易的。我们一次又一次地遇到这样的情形：电子工程师或者编程人员时常读过了半本书——《应用密码学》——然后就决定设计自己的系统。目前，我们还从未看到这样会产生一个像样的结果，并且我们在过去的十年中看到了很多的结果。假如一个学生学习了两个学期的土木工程课程之后，打算设计一座跨越白令海峡的具有革命性的新桥——在没有经过进一步的审查之前我们会实际建造并使用它吗？当然不会！然而，出于某些原因，当涉及到密码学时，人们却会给这种人支付工资并开发新手设计的系统。由于不正确的密码系统在遭受严重的攻击之前看起来很像是一个有效的密码系统，一些客户就会受到蒙骗并购买这些产品。

不要跌入陷阱，认为密码学很容易，实际并非如此！

2.8　易于实现的密码系统

即使密码系统本身是困难的，但它仍然是一个安全系统中比较容易实现的部分。如同锁一样，密码组件有良好定义的边界和需求。清晰地定义整个安全系统则困难得多，因为它涉及到更多的方面，诸如用于授权访问的组织程序、用于检查其他程序是否得到执行的程序，这一类问题很难处理，因为情况总是在变化。计算机安全中的另一个重要问题在于软件的低劣质量。如果机器中的软件包含了成千的 bug（缺陷）并导致安全漏洞，那么安全软件就不可能生效。

密码系统是容易实现的，因为有专门的人才可以完成相应的工作。有一些专家愿意为你设计密码系统。他们并不是廉价的，并且与他们一起工作是很辛苦的。他们通常坚持要改变系统

的其他部件以达到期望的安全性。同样，对于所有的应用目标来说，密码系统也带来了问题，然而我们知道该如何解决。

其他的安全系统包含了一些我们不知道如何解决的问题。密钥管理和密钥存储在任何密码系统中都是至关重要的，但是大多数计算机都没有一个安全的地方来存储密钥。我们根本不知道如何处理劣质的软件质量问题。网络安全同样难以实现。当你把用户加入进去之后，问题变得几乎难以解决了。

2.9 背景阅读材料

对密码学感兴趣的人应该阅读David Kahn的《破译者》(“The Codebreakers”) [46]。这是一部密码学的历史——从远古时代到20世纪。其中的故事提供了很多的例子，说明了密码系统工程师面临的问题。

接下来应该阅读Bruce的第一本书——《应用密码学》(“Applied Cryptography”) [88]。《应用密码学》涵盖了更多的内容，包括了它所讨论的所有算法的详细说明。然而，该书没有探究本书中讨论的详细的工程要点。

关于各种事实与精确的结果，没有比Menezes、van Oorschot和Vanstone的《应用密码学手册》(“Handbook of Applied Cryptography”)[65]写得更好的了。它是一部密码学的百科全书，一部非常有用的参考书，但是正像其他百科全书一样，很难通过这样的一部书掌握这一领域的全部技术。

Bruce早先所著的《秘密与谎言》(“Secrets and Lies”) [89]大体上对计算机安全以及如何将密码学用于该框架做了一个很好的解释。关于安全工程方面，没有比Ross Anderson的《安全工程》(“Security Engineering”)[1]更好的书了。这两本书对于理解密码学的背景都是必不可少的。

PDG

第3章 密码学简介

本章介绍密码学的基本概念以及阅读本书其他部分所需要的背景信息。

3.1 加密

加密是密码系统最初的目的。加密的一般框架如图3.1所示。Alice和Bob希望进行相互通信［这里使用的人名（尤其是Alice、Bob和Eve）都遵循密码系统的惯例］，然而通信信道一般是不安全的。Eve正在信道上进行窃听。对于Alice发送给Bob的任何消息m，Eve也能接收到（同样，Bob向Alice发送的消息也能被Eve接收到。但是，除了Alice和Bob的角色互换之外，这是相同的问题。只要能够保护Alice的消息，采用同样的解决方案也能保护Bob的消息，因此我们只关注Alice的消息）。Alice和Bob如何通信才能不让Eve得到任何内容呢？

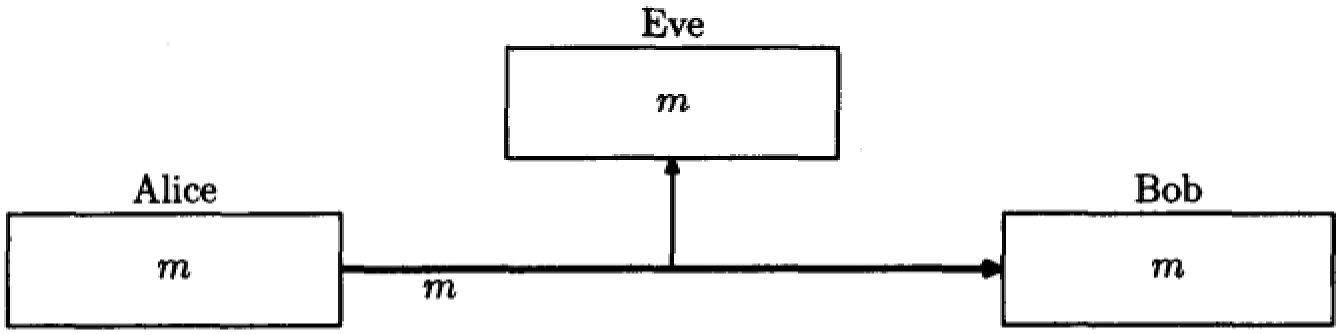


图3.1 Alice和Bob如何才能进行安全通信呢？

为了防止Eve获知Alice和Bob进行的会话，他们使用图3.2所示的加密方法。首先，Alice和Bob协商一个密钥K_e。他们必须通过某个Eve不能够窃听的信道来协商这个密钥。也许Alice给Bob邮寄密钥的一个副本，或者采用其他类似的方法。

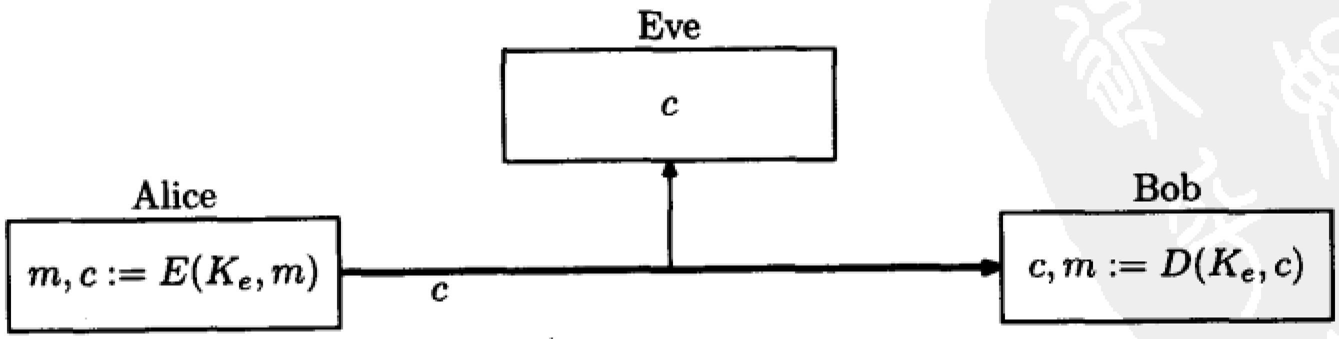


图3.2 一般的加密模型

当Alice想要发送消息m时，她首先利用一个加密函数加密这个消息。我们把加密函数记为$E(K_e, m)$，并把其结果称为密文c（原始消息m称为明文）。Alice发送密文$c := E(K_e, m)$给Bob，

而不是发送消息m。当Bob接收到c之后，他利用解密函数$D(K_e, c)$来解密密文，从而得到Alice想要发送给他的原始消息m。

但是，Eve不知道密钥K_e，因而当她得到密文c时不能够对密文进行解密。一个好的加密函数应该使他人在不知道密钥的情况下不可能由密文c找到明文m。

这个框架的一个明显的应用就是传送电子邮件，除此之外，它也可以用于存储。存储信息可以看做是以时间的方式而不是以空间的方式传送消息。在这种情况下，Alice和Bob通常是同一个人，但是同样的解决方案在这里也适用。

3.1.1 Kerckhoffs 原理

为了解密密文，Bob需要两样东西，即必须知道解密函数D和密钥K_e。这里，一个重要的准则就是Kerckhoffs原理：加密方案的安全性必须仅仅依赖于密钥K_e的保密性，而不依赖于算法的保密性。

这个准则之所以具有合理性是因为算法是难以改变的，它们被嵌入到软件或者硬件中，因而难以升级。在实际情况中，同一算法要使用很长一段时间，这就是现实。而且，仅仅保持密钥的秘密性是相当困难的，保持算法的秘密性就更加困难了（其代价也非常昂贵）。没人只为了两个用户而建立一个密码系统。系统中的任何参与方（可能以数百万计）都使用相同的算法。Eve只需要从其中一个参与方那里就可以得到这个算法，而其中必有一个参与方是容易收买的。或者，她可以窃取一台便携式电脑，获得其中的算法。还记得我们的妄想狂模型吗？Eve也可能恰好是这个系统中的其他用户，甚至是该系统的设计者。

还有许多理由表明算法应该是公开的。根据经验，我们知道人们很容易由于一个小错误而设计一个薄弱的密码算法。如果算法不公开，那么就没有人发现这个缺点，直到敌手设法攻击它为止。然后敌手就能够利用这个缺点来毁坏系统。作为顾问，我们已经分析了许多秘密的加密算法，所有这些都是较弱的算法。不要被这样的老观念愚弄了："噢，我们对算法也保密是为了增加安全性"。那是错误的。采用这样的做法来提高安全性的可能性是很小的，而降低安全性的可能性是极大的。保密的算法通常非常糟糕，以至于任何与之相关的断言几乎都是无能力的证明。

3.2 认证

在图3.1中，Alice和Bob存在另外一个问题。Eve能够做的不只是偷听消息，Eve能够以某种方式改变消息。这要求Eve对通信信道有更强的控制能力，但这不是不可能的。比如，在图3.3中，Alice试图发送消息m，但是Eve干扰了通信信道。Bob接收到的是另外一个消息m'，而不是消息m。我们假设Eve同时也获得了Alice试图发送的消息m。Eve能够做的其他事情包括删除一个消息以使得Bob根本接收不到它，插入一个她自己捏造的消息，记录一个消息并在以后发送给Bob，或者改变这些消息的顺序。

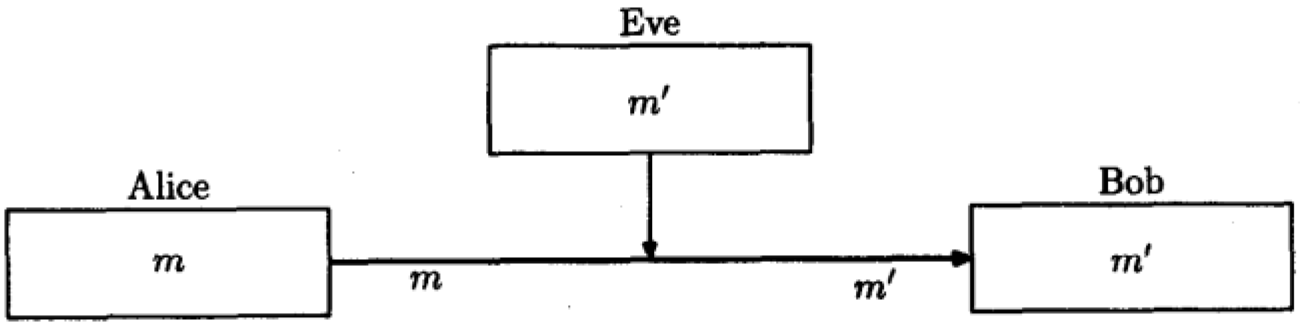


图 3.3 Bob如何知道是谁发送了消息？

从另外一个角度考虑：Bob刚刚接收到了一个消息。为什么Bob应该相信这个消息来自于Alice呢？他没有任何理由相信这一点。如果他不知道是谁发送了这个消息，那么这个消息也就毫无价值了。

为了解决这个问题，我们引入认证的概念。像加密一样，认证使用一个密钥，而Alice和Bob都知道这个密钥。我们称认证密钥为K_a，以区别于加密密钥K_e。图3.4给出了认证消息m的过程。当Alice发送消息m时，她通过计算确定一个消息认证码，或称MAC。Alice确定MAC a为$a := h(K_a, m)$，其中，h是MAC函数，K_a是认证密钥。现在Alice把m和a都发送给Bob。当Bob接收到m和a时，他利用认证密钥K_a重新计算a究竟应该是什么，并检查他接收到的a是否正确。

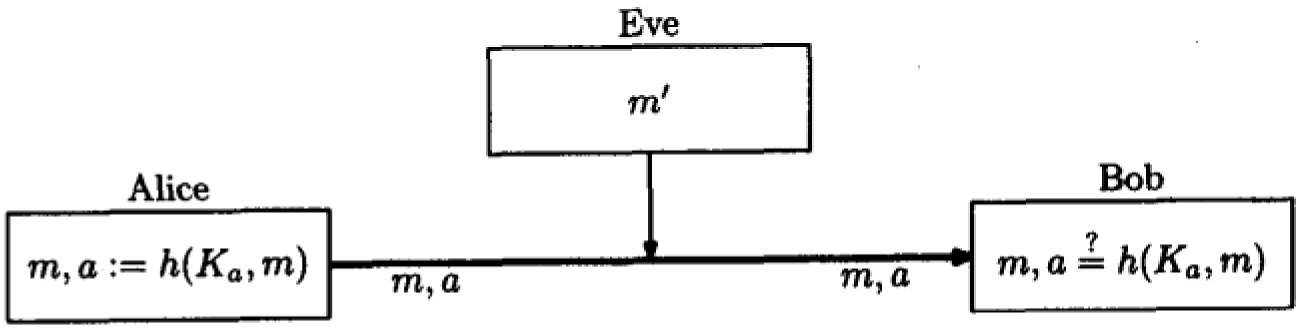


图 3.4 一般的认证模型

现在Eve想要把消息m更改为一个不同的消息m'。如果她只是把m替换为m'，那么Bob将计算$h(K_a, m')$，然后把它与a进行比较。但是，对于两个不同的消息，一个好的MAC函数不会输出相同的结果，于是Bob能够辨识出这个消息是不正确的。既然消息发生了错误，Bob就要丢弃它。

如果假定Eve不知道认证密钥K_a，那么Eve得到消息及其有效MAC的惟一方式就是当Alice给Bob发送消息时进行偷听。这仍然允许Eve做一些恶作剧。Eve能够记录一些消息及其MAC，随后进行重放并发送给Bob。

纯粹的认证只是部分解决方案。Eve仍然可以删除Alice发送的消息，她也可以重复旧消息或者改变消息的次序。因此，认证几乎总是和一个对消息顺序进行编号的编号方案结合在一起使用的。如果m包含这样一个消息编号，那么当Eve重放旧消息时Bob就不会被欺骗。Bob将清楚地看到该消息具有一个正确的MAC，但是其顺序编号却是一个旧消息的编号，因此他会丢弃它。

认证与消息编号方案结合起来使用可以解决大多数问题。而Eve仍然可以阻止Alice和Bob进行通信，或者通过先删除消息而随后再发送给Bob的方式来延迟消息传递。但她能够做到的也就是这些。

最好的办法是考虑这样的情况：Alice 发送一列消息 $m_1, m_2, m_3, \cdots$，Bob 只接收具有正确的MAC的消息，而且这些消息的编号严格大于①他最后一次接收消息的消息编号。从而Bob接收的一列消息是 Alice 发送的消息序列的子序列。子序列可以与原序列相同（没有消息被删除），也可以是原序列删除一些消息之后的序列。

在这种情况下，密码学能够提供一定的帮助。Bob 将接收到 Alice 发送的消息的一个子序列，但是Eve除了删除某些消息或者阻止所有通信之外，不能够操纵消息的传输。为了防止信息丢失，Alice 和 Bob 通常使用某种方案来重新发送那些丢失的消息，但是那些是针对特定的应用而言的，不属于密码学的范畴。

当然，在很多情形中，Alice 和 Bob 想要同时使用加密和认证。我们将在后面非常详细地讨论这种结合。绝不要混淆这两个概念。加密一个消息并不能阻止操纵其内容，而认证一个消息不能使该消息保密。密码学中一个常见的误解就是认为加密一个消息能够防止Eve改变它。事实并非如此！

3.3 公钥加密

要使用3.1 节讨论的加密方案，Alice和Bob必须共享一个密钥 K_e。如果他们相距很远，如何才能够共享一个密钥呢？ Alice 不能通过通信信道把密钥发送给 Bob，因为 Eve 也能够读取该密钥。分发和管理密钥的问题是密码学中一个真正困难的问题，对此我们只能解决其中的部分问题。

Alice 和 Bob 可能在上个月一起喝酒的时候交换过密钥。但是，如果 Alice 和 Bob 是一个群组的成员，该群组包括 20 个想要互相通信的朋友，那么群组中的每一个成员就必须总共交换19 个密钥。群组中的所有成员一共要交换 190 个密钥。这已经非常复杂了，而这个问题会随着与 Alice 通信人数的增加而变得更加复杂。

建立密钥是一个古老的问题，其中一个重要的解决方案是使用公钥密码学。我们首先讨论公钥加密，如图 3.5 所示。在这个图中我们不考虑 Eve。从现在开始，假定像 Eve 这样的敌手总是容易得到所有的通信。除了没有 Eve，这个图非常类似于图 3.2。一个很大的差别是 Alice 和 Bob 不再使用相同的密钥，而是使用不同的密钥。这就是公钥密码学的杰出思想。

为此，Bob 首先使用一个特殊的算法生成一对密钥（S_{Bob}, P_{Bob}）。这两个密钥是私钥 S_{Bob} 和公钥 P_{Bob}。然后 Bob 做了一件令人惊讶的事情：公开 P_{Bob} 作为他的公钥（否则怎能称之为公钥呢？）。

① “严格大于”意思是“大于并且不等于”。

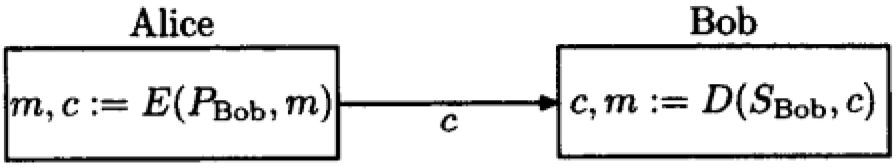


图 3.5　公钥加密的一般模型

当 Alice 想要发送一个消息给 Bob 时，她查寻 Bob 公开的公钥。Alice 使用公钥 P_{Bob} 加密消息 m 得到密文 c，把 c 发送给 Bob。Bob 使用他自己的私钥 S_{Bob} 和解密算法来解密密文 c 而得到消息 m。

为了使这个机制能够工作，密钥必须确保对生成算法、加密算法和解密算法来说解密确实能够产生原始消息。换句话说，$D(S_{\text{Bob}}, E(P_{\text{Bob}}, m)) = m$ 必须对于所有可能的消息 m 都成立。我们将在后面进行详细分析。

不仅 Alice 和 Bob 使用的两个密钥是不同的，而且加密算法和解密算法也是完全不一样的。所有的公钥加密方案都密切地依赖于数学。有一个明显的要求就是，从公钥中计算出相应的私钥应该是不可行的。

这种类型的加密就称为非对称加密，或者公钥加密，它与我们以前讨论过的对称加密或者私钥加密相对应。

公钥密码学使得分配密钥的问题变得非常简单。现在，Bob 只需要分发一个公钥，任何人都可以使用它。Alice 也用同样的方法公开她的公钥，然后 Alice 和 Bob 就可以安全地通信了。即使在大型的群组中，每一个群组成员只需要公开一个公钥，这就相当容易管理了。

这样看来，如果公钥加密如此简单易行，我们为何还要考虑对称加密呢？因为公开密钥加密的效率非常低，它们相差几个数量级。在所有情况下都使用公钥加密显然代价太高了。在使用公钥密码学的实际系统中，几乎总会看到公钥算法和私钥算法混合使用的情况。公钥算法用来建立一个秘密密钥，该秘密密钥用来加密真正的数据。这样就将公钥密码学的灵活性和对称密码学的高效性结合起来了。

3.4　数字签名

数字签名是消息认证码在公钥情形下的等价形式，其一般框架如图 3.6 所示。这次是 Alice 使用密钥生成算法来产生一个密钥对（$S_{\text{Alice}}, P_{\text{Alice}}$）并公开她的公钥 P_{Alice}。当她要发送一个带签名的消息 m 给 Bob 时，她通过计算确定签名 $s := \sigma(S_{\text{Alice}}, m)$。Alice 发送 m 和 s 给 Bob，Bob 利用 Alice 的公钥通过验证算法 $\upsilon(P_{\text{Alice}}, m, s)$ 来验证签名。签名的工作机制类似于 MAC，不同的是 Bob 可以利用公钥进行验证，而产生一个新的签名却需要私钥。

Bob 只需要拥有 Alice 的公钥就可以验证消息是否来自于 Alice。非常有趣的是，任何人都能够得到 Alice 的公钥从而验证该消息是否的确来自于 Alice，这就是我们称 s 为数字签名的原因。在某种意义上，Alice 签署了该消息。在任何时候，如果发生了争执，Bob 可以把 m 和 s 带给法官来证明 Alice 签署了该消息。

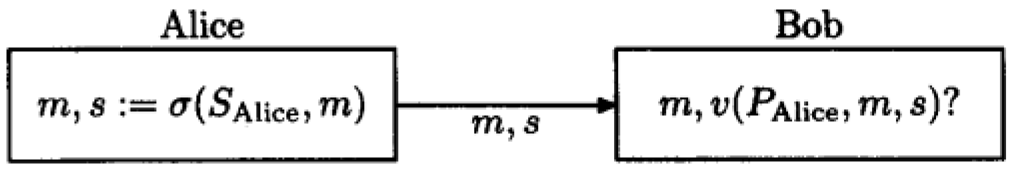


图 3.6 数字签名的一般模型

这在理论上是没有问题的，而且也很有效。然而在现实生活中，数字签名并不像我们希望的那样有用。主要的问题在于不是Alice本人亲自计算签名，而是她的计算机来计算签名。因此数字签名并没有证明Alice认可了该消息（即使在她的计算机屏幕上看到了该消息）。既然病毒能够轻易地控制计算机，数字签名实际上不能说明什么。然而，如果正确地加以使用，数字签名是极其有用的。

3.5 PKI

公钥密码学使得密钥管理变得简单，但是Alice仍然需要找到Bob的公钥。她怎样才能相信那是Bob的公钥，而不是其他人的公钥呢？也许Eve创建了一个密钥对，而在冒充Bob时公开该公钥。一般的解决办法是利用PKI，或者称公钥基础设施。

主要思想是使用一个中心机构，称为证书机构或者CA。每一个用户都把他的公钥报给CA并向CA证明自己，然后CA使用数字签名签署用户的公钥。签署的消息（或者称为证书）声明：“我，CA，已经查证公钥 P_{Bob} 属于Bob”。证书通常包括有效期以及其他的有用信息。

利用证书，Alice可以很容易找到Bob的公钥。我们假定Alice拥有CA的公钥并已经核实它是正确的公钥，Alice利用她已经拥有的CA的公钥检验关于Bob的密钥的证书。该证书保证了她拥有正确的公钥来与Bob进行通信。同样，Bob可以找到Alice的公钥，并且确信他正在与正确的人进行通信。

在PKI中，每一个参与方只需要让CA来证实他的公钥，并且要知道CA的公钥以便验证其他参与方的证书。这样的工作量远比与任意一个通信方交换密钥的工作量小很多。这就是PKI的巨大优势：一次注册，随处使用。

出于实际的考虑，PKI通常设置为多层次的CA。其中有一个高层的CA，称为根，它为低层CA的公钥颁发证书，而低层CA则证实用户的公钥。这个系统仍然以同样的方式运转，但是现在Alice需要检验两个证书以验证Bob的密钥。

PKI不是最终的解决办法，其中还有很多的问题。首先，CA必须为每一个人所信任。在有些情形下，这是容易做到的。一个公司的人力资源部门了解所有的员工，可以担任CA的角色。但是世界上没有一个实体能够取得所有人的信任。靠一个PKI控制整个世界的想法显然是错误的。

第二个问题是责任问题。如果CA颁发了一个错误的证书，或者CA的私钥被窃取了，那么该怎么办？Alice将信任一个错误的证书，并且可能因此而损失大量的金钱。谁来对此负责？显然，如果CA愿意以某种保险金的方式给予支持，Alice可以信任这些证书。这就需要Alice和CA之间建立一种非常具体的商业关系。

目前有很多公司都在试图成为世界性的CA，VeriSign大概是最为有名的一个。然而，对于它未能履行职责的情况，VeriSign明确地限定了自己的赔偿责任。在大多数情况下其赔偿限定在100美元，这远小于我们所支付的：利用VeriSign签署的证书进行安全保护的交易。这不是一个问题，因为信用卡支付对于消费者来说是安全的。然而，在下一次购买汽车时我们就不会再次使用仅返还100美元保证金的VeriSign的证书了。

有关PKI更详尽的信息以及相关问题，请参考第19章、第20章和第21章。

3.6 攻击

介绍完密码学的最重要的功能，现在我们谈一谈攻击。攻击有很多种类型，每一种都有其相应的严重性。

3.6.1 惟密文攻击

惟密文攻击就是大多数人谈到破坏一个加密系统时所指的攻击。在这种情况下，因为Alice和Bob对他们的数据进行了加密，攻击者得到的只是密文。在只知道密文的情形下试图解密一个消息就称为惟密文攻击。这是最为困难的一种攻击方式，因为你获得的信息最少。

3.6.2 已知明文攻击

已知明文攻击是一种既知道明文又知道密文的攻击。当然，这种攻击的目标是得到加密密钥。乍一看来这好像是难以置信的：如何知道明文呢？实际上，在很多情形中可以获得通信的明文。有时一些消息是容易预测的。比如：Alice在远处度假，她有一个E-mail自动回复设置，给每一封收到的E-mail发送“我在度假”的答复。通过给Alice发送一个E-mail并读取其答复，就可以得到该消息的一个原样副本。当Bob发送一个E-mail给Alice时，自动回复设置也同样答复，这一次是加密的。现在你就得到了一个消息的密文和明文。如果能够得到密钥，就可以解密Alice和Bob交换的所有其他消息。

另外一个典型的情况是，Alice发送同一个消息给多个人，包括你自己。现在你就得到了她发送给所有其他人的明文和密文副本。

也许Alice和Bob正在互相发送一个新闻稿的草稿。一旦该新闻稿被发布，你就得到了明文及其密文。

即使不知道完整的明文，也经常可以知道它的一部分。E-mail有一个可以预测的开端，或者结尾有一个固定的签名。一个IP包的包头是高度可预测的。通过这些可预测的数据可获得部分可知明文，我们把这些归类为已知明文攻击。

已知明文攻击比惟密文攻击更有效。作为一个攻击者，你可以得到比惟密文攻击更多的信息。这些额外的信息当然会对你很有帮助。

3.6.3 选择明文攻击

接下来的控制级别允许你选择明文。这是一类比已知明文攻击更强的攻击类型。现在你可以选择特殊的准备好的明文，这些明文的选取是为了更易于攻击该系统。你可以选择任意数量的明文，并得到相应的密文。需要再一次说明的是，在现实中这并不是不切实际的。在很多情形中，攻击者可以选择被加密的数据。Alice从外界的消息源（比如，可能受到攻击者影响的）得到信息，然后以加密的形式转发该信息给Bob，这是经常发生的。

选择明文攻击并非不合理。好的加密算法在抵抗选择明文攻击时应该没有任何问题。如果有人曾经试图以选择明文攻击不实际来说服你，你可以同他打赌以证明他的全新加密算法是如此脆弱，甚至不能抵抗选择明文攻击。

这种攻击有两个变形。对于离线攻击，在得到密文之前，可以准备一列想要加密的消息。对于在线攻击，可以根据已经得到的密文选择新的明文。大多数情况下可以忽略这种区别。我们通常讨论在线形式的攻击，这是两者中比较强的一个。

3.6.4 选择密文攻击

“选择密文”这一术语不太恰当。实际上它应该称为“选择密文和明文攻击”。在选择明文攻击中，可以选择明文值。在选择密文攻击中，既可以选择明文值也可以选择密文值。对每一个选择的明文，你可以得到相应的密文，而且对每一个选择的密文也可以得到相应的明文。

显而易见，选择密文攻击比选择明文攻击更强，因为攻击者有更多的自由。其目标仍然是恢复密钥。需要再一次说明的是，任何合理的加密方案都应该经受得住选择密文攻击。

3.6.5 区分攻击

上面描述的攻击是要恢复明文或者加密密钥。有一些攻击不需要恢复加密密钥，而是解密另外一个特定的消息。还有一些攻击不用恢复消息，而是泄露该消息的某些部分信息。现在已经有太多的攻击形式了，并且新的攻击形式还在不断被发明。那么我们究竟应该防御什么样的攻击呢?

最好的解决办法是定义区分攻击。区分攻击是一种非平凡的攻击方法，其目的是发现理想密码和实际密码之间的差异。这种攻击涵盖了迄今为止我们讨论的所有攻击，也包括了尚待发现的任何其他攻击。当然，我们必须定义什么是理想密码。

这样做是不是太不自然了?哦，不。我们的经验表明，你当然希望基本组件是完美的。一些加密函数的缺陷导致它们不满足区分攻击的定义，但这与说它们是完全符合要求的加密函数没有矛盾。每一次使用它们的时候，都需要检查这些缺陷以使其不会导致任何问题。在一个包括多个组件的系统中，也必须检查任何有缺陷的组合是否会导致一些问题。这很快就行不通了，而且在实践中，我们发现一些实际系统所展现出来的脆弱性都是由于其组件的缺陷造成的。

3.6.6 生日攻击

生日攻击以生日悖论这一诡辩而命名。如果一个房间里有23个人，那么他们中间有两个人的生日相同的可能性会超过50%。这个概率大得令人不可思议，因为一共有365个可能的生日。

什么是生日攻击呢？这种攻击依赖于这样一个事实：完全相同的值（也称为碰撞）的出现比你预料的要快得多。在一个保护金融交易的系统中，假设每一个交易使用新的64比特的认证密钥（为了简单起见，我们假定不使用加密），那么一共有 2^{64}（$= 18 \cdot 10^{18}$）个可能的密钥值，所以这应该是很难攻破的，对吗？不对！人们观察到大约 2^{32} 次交易之后，攻击者可以预期到两次交易使用了相同的密钥。假设第一个认证的消息总是相同的消息："Are you ready to receive a transaction?" 如果两次交易使用了相同的认证密钥，那么它们的第一个消息的MAC值也是相同的，攻击者很容易察觉到这一点。但是，知道了两次密钥是相同的，攻击者就可以把以前交易中的消息插入到正在进行的新交易中。有了正确的密钥认证，这些伪造的消息就会被接受，因此显然攻破了金融交易系统。

一般说来，如果一个元素有 N 个不同的取值，那么选择了大约 $\sqrt{N}$ 个元素之后，就预期有第一次碰撞出现。这里我们不考虑确切的细节，但是 $\sqrt{N}$ 是非常接近的。对于生日悖论来说，$N = 365$，从而 $\sqrt{N} \approx 19$。相同生日出现的概率超过50%所需要的人数实际上是23，但是对我们来说 $\sqrt{N}$ 已经充分接近了。作为考虑这件事情的一种方法，如果选择 k 个元素，那么有 $k(k-1)/2$ 对元素，每一对的两个元素有 $1/N$ 的机会是相等的。所以，找到一个碰撞的机会接近于 $k(k-1)/2N$。当 $k \approx \sqrt{N}$ 时这个可能性就接近于50%了①。

大多数时候我们讨论 n 比特的值。因为有 2^n 个可能的值，你需要 $\sqrt{2^n} = 2^{n/2}$ 个集合中的元素才能预期出现一个碰撞。我们将经常谈到这个 $2^{n/2}$ 界，或称为生日界。

3.6.7 中间相遇攻击

中间相遇攻击和生日攻击属于同一类（它们一同被称为碰撞攻击）。它们相当常见而且非常有用。你可以自己选择构造一个密钥表，而不是等待密钥重复出现。

我们回到前面的金融交易系统的例子，其中每一次交易都使用一个新的64比特的密钥来认证。通过使用中间相遇攻击，攻击者可以进一步攻破系统。他是这样做的：随机选择 2^{32} 个不同的64比特的密钥。对其中的每一个密钥，攻击者计算消息"Are you ready to receive a transaction?"的MAC值，并把MAC结果和密钥都保存在一个表中。然后他偷听每一次交易，并检查第一个消息的MAC是否出现在他的表中。如果MAC的确出现在表中，那么该交易的认证密钥将以很大的可能性与攻击者计算其表格的该条目使用的密钥是相同的，而这个密钥值与MAC值一起存储在表中。既然攻击者知道了认证密钥，就可以在交易中插入他任意选择的消息。（生日攻击只允许攻击者插入以前交易的消息。）

① 这些只是近似值，但这些值足以满足我们的要求。

攻击者需要偷听多少次交易呢？他预先计算了关于 $1/2^{32}$ 的密钥的MAC值，因此，无论何时系统选择一个密钥，都有 $1/2^{32}$ 的可能性使他能够识别出这个选择的密钥。于是经过 2^{32} 次交易之后，他能够期望有一次交易使用的密钥是他预先计算MAC时使用的密钥。攻击者总的工作量大约是 2^{32} 步预先的计算，加上偷听 2^{32} 次交易，比起尝试 2^{64} 个可能的密钥来，这种工作量小多了。

生日攻击和中间相遇攻击的不同之处在于，生日攻击等待一个值在相同的元素集合中出现两次。而在中间相遇攻击中，你有两个集合并等待这两个集合出现重叠。两种情况下，你都期望在尝试大约相同的元素个数之后找到第一个结果。

同生日攻击相比，中间相遇攻击更具灵活性。让我们以更加抽象的方式来考虑这一点。假设我们有 N 个可能的值。第一个集合有 P 个元素，第二个有 Q 个元素，因此一共有 PQ 对元素，每一对元素有 $1/N$ 的可能性相等。一旦 PQ/N 接近于1，就预期有一个碰撞。最有效的选择方式是 $P \approx Q \approx \sqrt{N}$。这正好就是生日界。中间相遇攻击可以提供额外的灵活性。有时，获得一个集合中的元素比获得另一个集合中的元素要容易一些，惟一的要求是 PQ 接近于 N。可以选择 $P \approx N^{1/3}$，$Q \approx N^{2/3}$。在上述例子中，攻击者可以对第一个消息建立一个由 2^{40} 个可能的MAC值组成的列表，期望在偷听了 2^{24} 次交易之后找到第一个认证密钥。

当我们从理论上分析一个系统被攻击的难易时，通常使用两个大小均为 $\sqrt{N}$ 的集合，因为这一般会将攻击者需要执行的步骤减到最少。我们需要进行更详细的分析来表明获得一个集合的元素是否比获得另一个集合的元素更困难。如果想要在现实生活中执行中间相遇攻击，则需要小心地选择集合的大小，以可能付出的最小代价来保证 $PQ \approx N$。

3.6.8 其他类型的攻击

到目前为止，我们主要讨论了对于加密函数的攻击。此外还可以定义其他密码函数的攻击，如认证、数字签名等，我们将在适当的时候加以讨论。

3.7 安全等级

付诸足够多的努力，任何密码系统都能够被成功地攻击。真正的问题是攻破一个系统需要做多少工作。要确定一个攻击的工作量，一种很容易的办法是把它与穷举搜索进行比较。如果一个攻击需要 2^{235} 个工作步骤，那么它就相当于对一个235比特的值进行穷举搜索。

我们经常谈论一个攻击者需要采取一定数目的步骤，但是至今也没有详细说明什么是一个步骤。这在一定程度上是偷懒，但也是为了简化分析。当攻击加密函数时，对给定的密钥，单独计算一个给定消息的加密就是一个步骤。有时一个步骤仅仅是在一个表中查找某些东西。它是可变的。但在所有情况下，一个步骤能够被计算机在很短的时间内完成。有时它可以在一个时钟周期内完成，有时需要100万个时钟周期，但是根据密码攻击所需要的工作量，100万

个时钟周期作为一个单独的因素不是十分重要。基于步骤分析的易用性远超过其内在的不确定性。我们总是可以做出更详细的分析来查明一个步骤相当于多少工作量。为了进行快速估计，总是假定一个单独的步骤需要一个时钟周期。

目前设计的任何系统事实上都需要128比特的安全等级。也就是说，任何攻击都至少需要2^{128}个步骤。目前设计的一个新系统如果成功的话，很有希望运转30年，并且自其最后一次使用之后，应该为数据至少提供20年的机密性。所以，我们的目标是为未来50年提供安全性。这是一个相当高的等级，但也有一些工作根据摩尔定律进行推断并把它用于密码系统。128比特的安全性是足够的[63]。我们还可以探讨100比特或者110比特，但是没有密码模块采用大约110比特的密钥，于是我们使用128比特。

这种安全等级的概念仅仅是近似的。我们只是估量了攻击者必须做的工作量，而忽略了像内存或者磁场系统的交互作用等因素。只是处理攻击者的工作量就十分困难了，而使这个模型变得复杂将使得安全性分析更加困难，而且会增加忽略关键点的可能性。因为利用简单而审慎的方法的代价相对较低，因此我们采用简单的安全等级概念。

3.8　性能

几乎每次设计系统时，人们都抱怨其速度和性能方面的代价。关于这个问题我们有以下的考虑。

首先，安全性不可能免费得到。想获得安全性，必须付出代价。如果负担不起，就不会得到可靠的安全性。道理很简单。现代的软件经过了很多周期来设计漂亮的三维视窗，但是很少或甚至根本没有功能上的改进。安全性是工业界的主要问题之一，然而很多软件开发者都不愿意在安全性上耗费与设计视窗同样的时间。

设计每一个系统时，我们都努力使其尽可能地高效。然而，到一定程度之后，任何性能上的提高的代价都变得及其昂贵。一旦偏离了安全性的大道，就必须进行大量的分析工作来确保不会使用一个脆弱的系统。这样的分析需要富有经验的密码学家，而聘请他们也是很昂贵的。对于大多数系统，比起设计和实现一个更高效的安全系统的开支来说，购买一台速度更快的计算机是更加便宜的做法。

对于大多数计算机，密码系统的性能是不成问题的。现代的CPU速度很快，足以跟得上它们所处理的几乎所有的数据流。比如，在1 GHz的奔腾III CPU上使用AES加密一个100 Mb/s的数据链接只需要20%的周期（实际上更小，因为在链接上从不会达到100 Mb/s的传输速率，这是由于通信协议的开销的限制所致）。如果等上几个月购买机器，以同样的价格可以买到1.2 GHz的CPU，同1 GHz的CPU相比，这种开销变成零。一家公司曾经抱怨加密1 Gb/s的代价太大。处理数据的机器价值50多万美元，可以料想他们当然有能力购买一个额外的CPU或者在芯片中增加一个硬件以处理密码系统。

在有些情形中，密码系统会造成性能上的瓶颈。一个很好的例子是使用非常多的SSL连接的Web服务器。SSL连接的初始化使用了公钥密码系统，在服务器端需要大量的计算能力。的确，我们能够开发一个使服务器端效率更高的SSL替代者。可是，我们不会那样做。和支付给开发一个新协议的专家的费用相比，购买一些硬件加速器来处理现有的SSL协议更廉价。开发新协议还要支付使之标准化的费用，然后还要在服务器端和客户端实现它，而且要说服每一个人升级到新的浏览器版本。

最近偶然遇到一个很好的论坛，该论坛是要说服人们选择安全性而不是性能。"已经有足够多的不安全的快速系统了，我们不需要另外一个。"这是非常正确的。人们在安全性上采用折中办法的代价和尽力做好付出的代价几乎相同，而前者只能提供很小的安全性。我们认为，对安全性来说，要么做好，要么别做。

3.9 复杂性

不存在安全的复杂系统。

设计准则1： 复杂性是安全性最大的敌人。

这是一个简单的规则，但是真正理解它需要一点时间。IT行业在创建复杂系统方面实在是令人沮丧，他们只是开发某些软件，然后投入使用……大多数时间是标准在起作用，几乎没有软件能经得起推敲。

这与几乎到处使用的"测试-修补"的开发方法有关系：创建一些东西，测试错误，修补错误，再测试并发现更多的错误，等等。测试、修补、再重复。这个过程一直进行，直至公司指示产品必须投放市场或者直到程序员厌烦了为止。当然，这样的结果会运转得相当好，只要它只用于所测试的环境。这对于功能性来说可能已经足够好了，但是对于安全系统而言完全不合适。

这种测试-修补方法的问题在于，测试只是说明了错误的存在，实际上这些错误正是测试者所寻找的。而安全系统即使在遭受了那些机智并怀有恶意的人的攻击后仍然能够工作。一个系统不可能对于所有可能受到的攻击情形都进行测试，系统必须从一开始就是安全的。

让我们考虑下面的类比。假设你使用一种流行的编程语言编写一段中型的应用程序，你要修补其中的语法错误直至通过第一次编译，然后，没有经过进一步的测试就把它交付给顾客了。我相信没有人愿意使用以这种方式开发的产品。

但是，安全系统通常就是这样完成的。它们不可能得到测试，因为没有人知道如何进行测试。如果产品有错误（bug），那么这样的产品就是有缺陷的。因此，获得安全系统的惟一方式就是从一开始就构建一个非常健壮的系统。这就要求系统是简易的。

我们所知道的使得一个系统简易化的惟一方式就是把它模块化。从软件开发的角度，我们都知道这一点。但是这一次我们不能使其有任何bug，所以在模块化的时候必须不留情面。设计准则2是我们给出的主要原则。

设计准则 2：正确性必须是一个局部性质。

换句话说，不管系统的其他部分如何工作，系统的每一部分都应该正确运转。的确，我们不愿意听到“这不会是一个问题，因为系统的其他部分绝不会让它发生这个问题”。其他部分可能有bug，或者在将来的某一版本中会发生变化。系统的每一部分都应该对自己的功能负责任。

你将看到这一准则在本书中以多种形式得到应用。我们的许多选择要比当前正在使用的许多其他设计保守得多，这就是其中的原因之一。

PDG

第一部分

消息的安全性

第4章　分组密码

第5章　分组密码模式

第6章　Hash 函数

第7章　消息认证码

第8章　安全信道

第9章　实现问题（Ⅰ）

第4章 分 组 密 码

分组密码是密码系统的“构件”之一，也是目前密码学领域研究较为深入和完善的部分，有关的文献也很多。

4.1 分组密码简述

分组密码是定义在长度固定的明文块上的加密函数。在分组密码中，密文块与明文块的大小相同，称为分组密码的分组（块）长度。目前分组密码明文块的长度一般为128比特（16字节），它对128比特的明文块加密，生成128比特的密文。分组密码是可逆的，即存在一个解密函数（算法），可以由128比特的密文得到128比特的原明文。

为了完成对消息的加密，我们需要一个密钥，没有这样的密钥，就无法实现对消息的加密。密钥与明文和密文一样也是一个比特串，其长度（称为密钥的大小）一般为128比特或256比特。为了方便，我们用$E(K, p)$或$E_K(p)$表示用密钥K对明文p的加密，用$D(K, c)$或$D_K(c)$表示用密钥K对密文c进行解密。

分组密码主要用于信息的加密。对较短的消息可直接用分组密码进行加密，但对比其分组长度还长的消息，加密需要使用分组密码模式，这将在第5章叙述。

根据Kerckhoffs原理，我们总是假定加密和解密算法是公开的。这个观点一度很难被一些人接受，他们想对算法也保密。我们认为，永远不要相信保密的分组密码（或其他保密的密码基元）。

将分组密码看成是一个依赖于密钥的明文与密文对照表有时很有用。对给定的密钥，这个表定义了从明文到密文的映射。当然，这个表可能十分庞大，例如，对分组长度为32比特的分组密码来说，对应的表大小为16 GB；当分组长度为64比特时，表的大小为1.5×10^9 TB；若分组长度是128比特，则表大得已无法命名，为5×10^{39}字节。因此，在现实环境下建立这样的表是不切实际的，但明文与密文对照表仍然是分组密码的很有用的概念模型。分组密码是可逆的，这要求在明文与密文对照表中没有相同的项，即任意一个可能的密文值在表中只出现一次，否则解密函数就不能正确解密以得到惟一的明文。这个表实际上是所有可能值的一个重新排序，数学家们称之为置换，因此分组长度为k的分组密码也就是为每一个密钥指定一个定义在k比特长度值上的置换。

4.2 攻击类型

当给出了分组密码的定义后，分组密码的安全性似乎很简单：即确保明文是秘密的即可。这是分组密码的要求之一，但仅此一个要求是不够的。定义实际上仅要求分组密码在惟密文攻

击下是安全的，即在攻击者只知道密文的情况下是安全的。在已公开的攻击算法中，仅有少量的惟密文攻击[54，92]，而大多数属于选择明文攻击。我们在3.6节中已对一般的攻击形式做了概述。所有这些攻击方式都可用于分组密码，而且有几种攻击是分组密码特有的。

相关密钥攻击是其中之一，它由Eli Biham在1993年提出[7]。这种攻击假定攻击者可以访问几个加密函数，攻击者不知道每个加密函数的密钥，但他知道这些密钥之间的关系。这种攻击方式似乎很奇怪，但这类攻击很有用[50]，因为确实有些密码系统使用满足已知关系的不同密钥。一个专有系统可以通过将密钥加1为每个消息生成不同的密钥，这使连续的消息被连续数值的密钥加密。研究表明，这类密钥之间的关系可以用于攻击分组密码。

还有一些更奇妙的攻击类型。设计双鱼（Twofish）分组密码的研究小组提出了选择密钥攻击的概念，在这种攻击方式中，攻击者先指定密钥的某一部分，而后对密钥的其余部分完成相关密钥攻击[86]①。

我们需要研究如相关密钥攻击和选择密钥攻击这样一些攻击类型，理由如下：我们已经看到，在实际分组密码系统中，相关密钥攻击是很可能出现的，因此这样的攻击并不冷僻。分组密码是十分有用的素材，但有滥用的倾向。用分组密码构造Hash函数的标准方法是Davies-Meyer构造[96]。在Davies-Meyer构造的Hash函数中，攻击者选择分组密码的密钥进行相关密钥和选择密钥攻击。如果忽视了这些攻击类型或其他攻击类型，任何分组密码安全性的定义一定是不完整的。分组密码应该是一个拥有简单接口的模块，最简单的接口应保证具有人们期望分组密码所应有的所有性质。分组密码的不完整性会使系统各部分之间存在依赖关系，这极大地增加了使用这个分组密码的系统的复杂性。

4.3 理想的分组密码

为了定义分组密码的安全性，我们首先要定义理想的分组密码。什么是理想的分组密码呢？它应该是一个随机置换。更准确地说，我们期望对于每一个密钥，分组密码都是一个随机置换，而且不同的密钥所对应的置换应该是完全独立的。正如前面所述，你可以认为一个128比特的分组密码（即一个在128比特长度值上的置换）就是一个有2^{128}个元素而且每个元素均为128比特长的表。一个理想的分组密码由每个密钥所对应的置换表组成，而且这些置换从所有可能的置换集合中完全随机地选取。

严格来说，理想的分组密码的这个定义是不完整的，因为在定义中并没有确定置换表的选择方法。然而只要我们确定了置换的选择方法，这个理想的分组密码就不再是随机的了。因此，为了形式地定义分组密码，我们不能只考虑单个理想的分组密码，必须将理想的分组密码当做在所有可能的分组密码上的均匀随机分布。于是，只要提及理想的分组密码，则一定与概率相关。这是数学家们喜欢的处理方式，但它使我们的解释变得十分复杂，因此我们仍然采用随机选取分组密码这个相对简单的概念。

① 稍后的分析说明，这种攻击不适用于Twofish[32]，但也许能攻击其他分组密码。

4.4 分组密码安全性的定义

分组密码安全性的定义有很多(参见[53]),这些定义完全是数学上的,没有一个能描述我们所要讨论的所有内容,因此我们喜欢用一个简单的非正式定义。

定义1:*如果不存在攻击方法,则分组密码是安全的。*

这个定义有一点“重复”,因此我们需要定义分组密码的攻击。

定义2:*分组密码的攻击是一个区分分组密码与理想分组密码的非平凡的算法。*

区分一个分组密码与理想分组密码,意味着将给定的分组密码X与一个具有同样分组大小和密钥长度的理想分组密码进行比较。若将分组密码X或理想分组密码作为黑盒(black-box)函数,则有一个称为区分器的算法可以借助于这个黑盒函数区分它们(所谓的黑盒函数是指区分器可以用此函数计算其输出,但不了解函数本身)。区分器通过访问作为黑盒函数的加密函数和解密函数并随意选择密钥进行加密和解密,从而可以确定这个黑盒函数是分组密码X还是理想的分组密码。区分器不一定每次都能正确区分,但其成功区分的次数应远大于失败的次数。

当然,有一些区分算法是平凡的。我们可以用密钥0对明文0进行加密,看其结果是否与我们期望从分组密码X中得到的结果相匹配。这是一个区分器,若要使它成为一种攻击形式,则这个区分器一定不是平凡的。由于我们不能形式化地定义什么是平凡的区分器,因此使得定义分组密码的安全性变得很困难。这有点像对不文明的行为下定义,我们不能给出一个精确的定义,但我们能知道哪些行为是不文明的①。对一个区分器来说,如果我们可以为几乎所有的分组密码找到类似的区分器,那么这个区分器就是平凡的。我们可以为任何分组密码(甚至包括理想的分组密码)构造与前面定义的区分器类似的区分器,因此这个区分器是平凡的。

我们还可以构造更复杂的、平凡的区分器:分别用密钥$1, \cdots, 2^{32}$对明文0加密,计算密文的前32比特所对应的值出现的次数。假设我们发现在分组密码X中密文t出现了5次而不是期望的1次,这个性质对理想的分组密码不可能成立。我们可以为任意的分组密码X构造类似的区分器,因此这也是一个平凡的区分器。

如果我们按如下的方式定义一个区分器,事情会变得更复杂。首先列出密码系统可以计算的1000个统计数字,并计算密码系统X的每一项,我们希望以1/1000的概率找到一个统计数字,然后由其中最重要的统计数字构造一个区分器。当然,我们可以用同样的技术为任意的分组密码构造区分器,因此这也是一个平凡的区分器,所不同的是这种平凡性不仅依赖于区分器本身,而且还与如何找到这个构造区分器有关。这也就是为什么人们不能形式化地定义平凡性

① 1964年,美国高等法院法官Potter Stewart在定义“不文明”(“obscenity”)时说,“今天我不打算对此进行更仔细的定义……但我知道我所看到的是不是‘不文明’的行为。”

和分组密码安全性的原因。对分组密码的安全性，我们很想给出一个简明的定义，但目前人们对密码学的认识还不足以对此进行恰当的定义，而一个正式的但忽略了一些攻击的定义将会误导人们对安全性的认识。

我们还不能忘记要在安全性的定义中对区分器的计算能力做一定的限制，然而此举将导致问题进一步复杂化。如果分组密码的安全性指标是 n 比特，那么区分器算法应该比在 n 比特上的穷举搜索更有效，更直观地说就是区分器的计算量应小于 2^n。如果没有给出安全性指标，其安全性的强度应等于密钥的长度。有些区分器仅仅给出一个概率意义上的结论，这更像部分密钥搜索，这些区分器需要在计算量和区分能力之间进行权衡。例如，穷举搜索一半的密钥空间的计算量为 2^{n-1}，得到正确结论的可能性为75%（如果攻击者找到了密钥，他便得到了正确的结果；如果攻击者没有找到密钥，他仍有50%的机会猜到结果。总体来说，他有 $0.5 + 0.5 \times 0.5 = 0.75$ 的机会得到正确的结果）。将区分器与部分密钥搜索进行比较，我们要考虑计算量和区分能力之间的自然权衡，而不是把部分密钥搜索作为攻击方式进行分类。

我们的定义包括了所有可能的攻击方式：惟密文攻击、已知明文攻击、（动态）选择密文攻击、相关密钥攻击以及构成非平凡区分器的所有其他攻击，因此我们喜欢用这个定义。

为什么我们要用大量的篇幅来讨论分组密码安全性的定义呢？因为安全性的定义非常重要，它在分组密码与系统的其他部分之间定义了一个简单而清晰的接口，而这种模块化是杰出设计的特点。在一个安全系统中，复杂性是我们要考虑的重要因素，因此良好的模块化设计显得比在其他领域更重要。如果一个分组密码符合我们的安全性定义，那么可以将它当做理想的分组密码来处理。假如系统中的分组密码表现出了与理想的分组密码不同的特征，那么你就为它找到了一个区分器，这也就意味着这个分组密码不符合我们的定义。当使用一个安全的分组密码时，我们就认为它具有我们期望的所有性质，不再考虑它的特殊性和缺点。由于理想的分组密码是一个十分简单的概念，从而分组密码的设计也就变得更简单了。

4.4.1　置换的奇偶性

很遗憾，我们还有更复杂的问题。根据我们前面的说明，用一个确定的密钥进行加密对应于置换表中的一个查找。置换表的建立可分为下面两步：首先进行初始化，使得表中第 i 个位置上放置 i 值，然后通过多次交换表中两个元素的位置创建所要的置换表。可以证明所有的置换表分为两类：一类是经过偶数次交换元素位置得到的置换，称为偶置换；另一类是经过奇数次交换元素位置得到的置换，称为奇置换。两类置换的数量都恰好是所有置换的一半。

现代分组密码的分组长度为128比特，运算字长为32比特，已经证明这是一种非常成功的方法，但也有不好的一面：难以得到奇置换，从而使分组密码实际上只有偶置换。

这个性质使得我们可以得到一个几乎对所有分组密码都有效的区分器，我们称之为奇偶攻击。对给定的密钥，可以通过加密所有可能的明文获得这个置换。如果这个置换是奇置换，则可知这是理想的分组密码；如果是偶置换，则可判定是一个真实的分组密码，而且判断正确

的可能性为75%。显然，这个区分器只有在给定的是理想分组密码而且又得到了偶置换的情况下才可能出错，我们可以通过对多个密钥重复进行上述工作来提高正确率。

当然，这个攻击没有任何实际意义，因为要确定置换的奇偶性，必须计算加密函数的除一个明文密文对以外的其他所有明文密文对（最后一个可以由此导出），实际上并不允许获取如此多的明文密文对，其他的攻击可以比它更快地获得成功。特别是攻击者一旦获取了大多数明文密文对，他将不再需要密钥去解密给定消息，而可以简单地通过查询由明文密文对创建的表得到所要的结果。

我们可以勉强认为奇偶攻击是平凡的，但我们更倾向于对理想分组密码的定义进行适当的修改，使其只是随机地选择偶置换。

定义 3： 理想的分组密码实现为每个密钥独立地选择一个随机偶置换。

很可惜，该定义使理想的分组密码变得难以理解，而惟一的选择是认为几乎所有已知的分组密码都不合格。对绝大多数应用来说，将其限制在偶置换上是没有意义的，实际上只要不允许计算所有的明文密文对，我们就不能区分偶置换和奇置换。

如果你曾经拥有一个能产生奇置换的分组密码，可以回到原来的定义。实际上，奇偶攻击的意义主要在于安全性的形式定义而不在于实际的系统，因此可以忘掉有关奇偶攻击的结果。

4.5 实际的分组密码

目前，人们已经提出了几百种不同的分组密码。一般来说，设计一个新的分组密码非常容易，但要得到一个性能良好的分组密码却很困难。不用说，一个分组密码首先必须是安全的，声称一个分组密码是安全的就好比说一个新屋顶不漏雨，或者说新车有前灯一样，重要的是要求分组密码在各种不同的应用环境下是高效的。

设计新的分组密码是一件有趣和益智的事情，但在其他专家对此进行深入细致的研究之前，我们不应该相信它，切记：在实际系统中不要使用未知的分组密码。对一个新的分组密码来说，公开发表它是必须的，但仅此还不够，有很多分组密码并没有得到深入的研究，因此最好是使用已经进行过充分研究的分组密码。

实际上，所有的分组密码都是由一个弱的分组密码重复几次构成的，每一次重复称为一轮。这种由几个弱的轮次构成一个强的分组密码的结构易于设计和实现，而且也便于分析。大多数针对分组密码的攻击都是从减少了轮次的情形开始的。随着攻击技术的改进，更多轮次的系统也可以被攻击。

下面我们将详细地分析几个分组密码，但不可能面面俱到，我们主要关注其整体结构和性质，更详细的说明可以在一些参考书和因特网上找到。

4.5.1 DES

DES的密钥长度为56比特，分组长度为64比特，已不适应于现代的快速计算机和庞大的数据量，因此DES作为密码学中最辉煌的分组密码最终走向了衰落而退出了历史舞台[70]。另

外一种分组密码 3DES[73]由 3 个 DES 加密序列组成，是 DES 的另外一种形式。它解决了 DES 的小密钥问题，但没有解决小的分组长度问题。按现在的标准来说，DES 的速度还不够快，而 3DES 的速度仅为 DES 的 1/3。目前的很多系统仍然使用 DES，但我们建议在新的系统中不要使用 DES 和 3DES。

图 4.1 是 DES 中一轮加密的示意图，在其他文献中可以找到类似的图，它直观地显示了 DES的一轮运算过程。图中每一个方框代表一个特殊的函数，而带箭头的线表示该函数的输入数据，"⊕" 代表 XOR（异或）运算，有时又称为按位加法或不带进位的加法。

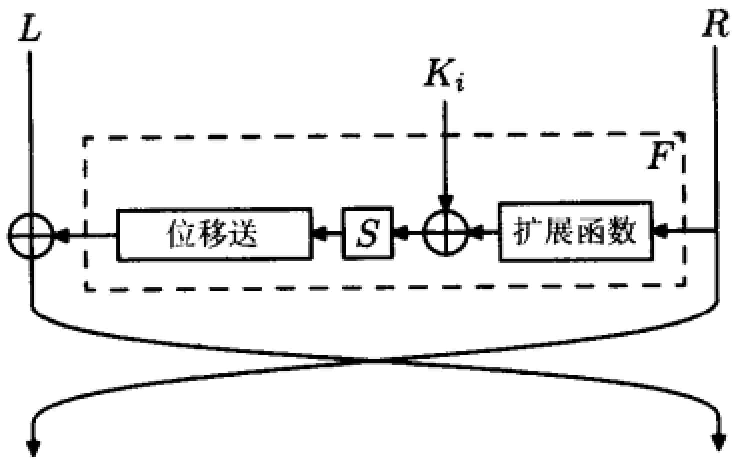


图 4.1　DES 单轮结构

DES 首先将长度为 64 比特的明文的每一比特位用"半有序"的方式重新排列（即对明文做一个初始的置换），并将其分为两个 32 比特的块，按位置左边的一半记为 L_0，右边的一半记为 R_0，然后进行加密运算，在加密的最后用初始置换的逆置换作用得到 64 比特的密文。这个初始的置换并没有密码学上的意义，也许没有人能知道 DES 的设计者为什么要这样做。

DES 共有 16 轮，每一轮都有一个 48 比特长的轮密钥。在轮密钥 K_i 的控制下，第 i 轮由（L_{i-1}, R_{i-1}）得到（L_i, R_i），这主要由轮函数 F 完成，如图中的虚线框所示。首先用一个扩展函数复制若干比特从而将 32 比特的 R_{i-1} 扩展为 48 比特，然后与 K_i 做异或运算，结果经过称为 S 盒的变换后变为 32 比特，再用一个置换作用后与 L_{i-1} 做异或运算得到 L_i，而 $R_i = L_{i-1}$。S 盒是一个公开的对照表，由 8 个将 6 比特的串映射为 4 比特的串的较小的表构成，从而实现从 48 比特到 32 比特的变换。

DES 的基本结构称为 Feistel 结构[30]。这种在每一轮中完成相同的操作［即 $L_i = R_{i-1} \oplus F(L_{i-1}, K_i)$，$R_i = L_{i-1}$］的想法很好，它保证了对加密和解密进行完全相同的操作，从而能比较容易地得到同时实现加密和解密的函数，而且还简化了分析工作。在完成最后一轮后，左右两半不再交换位置，这个小技巧进一步使得除了采用的轮密钥的顺序不同外，加密和解密函数完全相同，从而可用相同的电路来实现，这特别有利于算法的硬件实现。DES的密钥方案是：采用不同的选择方法，DES 为每一轮加密从 56 比特的系统密钥中选择 48 位作为轮密钥①。

① 关于密钥选择方案，请参阅 DES 说明书[70]。

DES的各个部分有不同的功能。Feistel结构简化了设计，而且使得左右两半混合在一起；与轮密钥的异或保证了密钥和数据的完全混合（mixed）；S盒提供了非线性，如果没有这种非线性，加密可以表示为一系列的异或运算，从而会发生一个非常简单的基于线性代数的攻击；最后，S盒、扩展函数以及置换保证了DES的扩散性（diffusion），由此使得如果轮函数输入有1比特发生了改变，则其输出的多个比特都将发生变化，从而在下一轮将会有更多的比特发生改变。如果没有良好的扩散性，必将导致明文的小改变只会引起密文的小变化，这很容易被检测出来。

按照我们对安全性的定义，DES具有一系列不满足安全性定义的性质。由于轮密钥是完全从系统密钥选择48比特得到的，若系统密钥为0，则所有的轮密钥都是0。另一方面，由于解密与加密的不同仅在于使用轮密钥的顺序不同，而此时所有的轮密钥都相同，因此密钥0所对应的加密函数和解密函数就完全相同。理想的分组密码没有这个性质，它会导致一个简单而有效的攻击[①]。

DES的另外一个性质是互补性，即对任意的密钥K和明文P，$E(\overline{K}, \overline{P}) = \overline{E(K, P)}$成立，这里$\overline{X}$表示对$X$的每一位求补，称为$X$的补。换句话说，用原密钥的补作为密钥对原明文的补进行加密，则得到的密文一定是原密文的补。

这一性质从图4.1很容易看出。设想将L_{i-1}, R_{i-1}, K_i改为$\overline{L_{i-1}}, \overline{R_{i-1}}, \overline{K_i}$，由于扩展函数仅仅是添加一些比特副本，因此扩展函数的输出也改为原输出补，再与$\overline{K_i}$做XOR运算后，结果一定保持不变，从而S盒的输出也不变，这样最后参与XOR运算的双方一个求补，一个保持不变，于是最后得到$\overline{L_i}, \overline{R_i}$。这说明每一轮的运算都有互补性，从而整个密码系统也具有这个性质。

理想的分组密码不具有这个奇特的性质，利用该性质可以攻击使用DES的系统。

总体来说，DES根本不能满足要求，它的密钥长度是不够的，目前存在几种通过简单的穷举搜索寻找DES密钥的方法。

3DES有更长的密钥，但它继承了DES弱密钥和互补性的性质，而按我们的定义，其中任意一个性质都使得它是不合格的。另一个缺陷是它的分组长度为64比特，这限制了我们用一个固定密钥加密数据的数量（详细的讨论见5.8节）。由此可见，3DES不是理想的分组密码，如果由于一些历史原因必须使用3DES，需要格外小心。

4.5.2 AES

高级加密标准（AES, Advanced Encryption Standard）是新的美国政府标准。美国国家标准和技术研究所（NIST）在1997年发起了在全世界范围内征集新的加密标准算法的活动，在众多的应征算法中有15个算法首轮入选[72]，其中有5个算法进入最后的决赛[74]，最终Rijndael算法胜出，成为新的加密标准算法AES。总体来说，从开始征集到最后确定标准算法的整个过程比预期的要顺利。对那些想对新的密码系统进行标准化的人来说，这种方法值得借鉴和学

① 另外还有3个具有这种性质的密钥，它们统称为DES的弱密钥。

习。要得到一个好的方案，采用AES式的竞选远比由某个团体设计要好。如果没有足够多的专家保证能得到一个好的方案，也许根本不应该进行标准化方面的工作。

AES的结构与DES不同，它不是Feistel结构的密码。图4.2是AES单轮流程图，其余的各轮与之类似。明文的长度为16个字节，在图中的最顶端，首先将其与16个字节的轮密钥（图中在“⊕”的侧面）进行异或运算，结果再按字节输入将8比特映射为8比特的S盒，所有字节对应的S盒都相同；然后将这16个字节用一个指定的顺序重新排列；最后分为4个组分别用线性混合函数进行混合，这里的线性指的是输出是几个输入的异或。

这是一轮的运算过程，得到的16个字节的输出作为下一轮的输入，而整个加密运算根据密钥的大小由10~14轮组成。与DES一样，AES也有一个轮密钥生成方案，但其结构却与DES的不同。

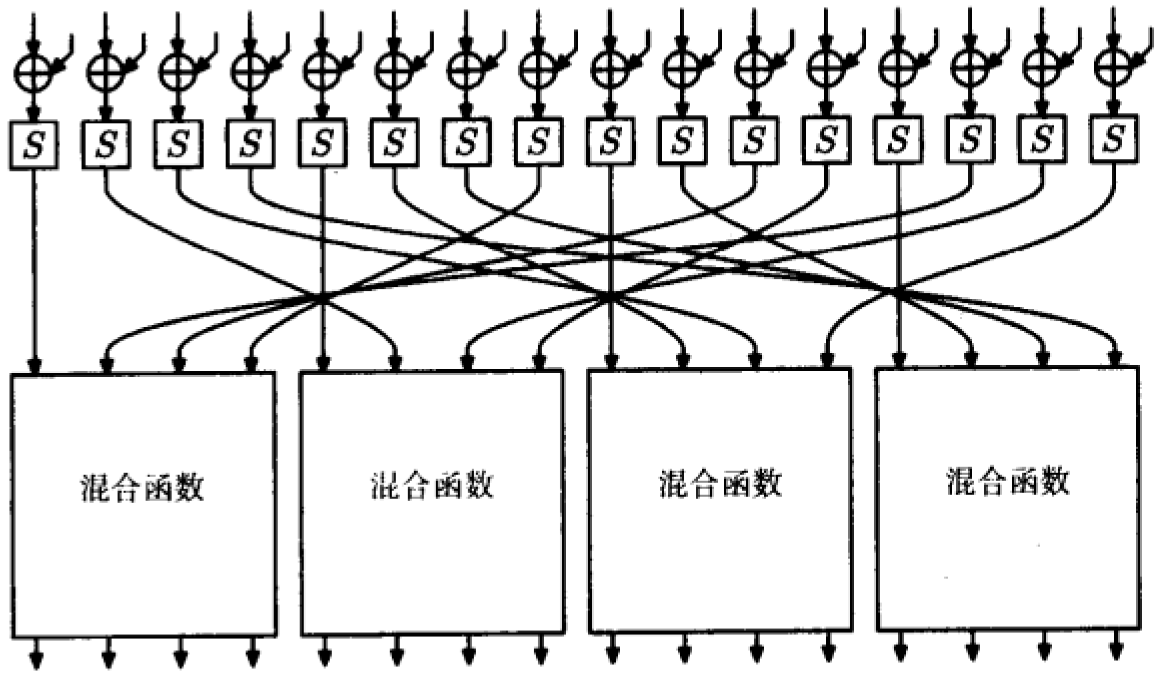


图4.2 AES单轮结构

AES的结构有一些好的方面，但也有一些不足。一方面，AES的每一步针对字节的运算都可以并行地完成，从而便于算法的高速实现。但另一方面，它的解密运算与加密运算有很大的差别，解密时需要S盒的逆，而且计算混合函数的逆与计算混合函数本身也不同。

我们可以发现AES中有与DES中相同的功能块，如与轮密钥的异或运算，由S盒提供的非线性，由字节置换和混合函数保证的扩散性。AES的结构非常清晰，每一部分都完成一个独立的任务。

AES是一个很具进取意识的设计，但我认为并不能完全相信AES的安全性。AES的设计者在他们提交的原始报告中给出了一个对6轮算法的攻击，并选择10~14轮的完整的密码算法，具体可由密钥的大小确定[18]。后来这个攻击方法有所改进，可以对密钥为128比特的7轮算法、密钥为192比特的8轮算法、密钥为256比特的9轮算法实施攻击[31]，因此，10~14轮的算法具有3~5轮的安全余地。从另一方面看，当密钥为128比特时，我们所知道的最有效的攻

击方法可以攻击70%的密码系统，换句话说，AES的安全性寄希望于未来的攻击不会有较大的改进。

未来常常是不能预测的，但有时回顾过去对我们很有帮助。到目前为止，人们分析的最为彻底的密码系统有DES、FEAL和IDEA。一般来说，在算法公布多年后，攻击算法才会有重大的改进。虽然在这个领域的研究已经取得了进步，但仍然很难使人相信我们已经完全掌握了所有技术，并且很难使人相信攻击方法将不会有重大的改进。

这主要是密码学家们关心的事，即使现在的攻击方法已经改进到可以攻击AES的程度，它也可能是需要2^{120}步的运算和2^{100}字节的内存的攻击方法。按照我们的标准，这足以说明密码系统已被攻破，从而要求我们将安全性水平降低到120比特。这将不再满足我们的要求，但是如果我们的财产受到威胁，我们也不必对此感到担忧，这样的攻击并没有实际意义，在未来的50年之内也不可能成为实际的攻击（见3.7节）。

AES令人担忧的是它具有简单的代数结构[34]。AES的加密可以表示为具有256个元素的有限域上相对简单的封闭的代数公式，这仅仅是一种表示，不构成一个攻击，但是如果有人能求解这些公式，则AES将被攻破。这将开辟新的攻击途径。据我们所知，其他的分组密码都没有这样简单的代数表示，我们现在还不知道AES这种表示是否能导致对它的攻击，因此有足够的理由对AES的使用产生疑虑。我们设想再过5年，当AES得到广泛应用时的一个可怕的情形：我们正坐在一个大房间里听一个数学专业的博士生的报告，“一天下午我感到很无聊，看了看我朋友的密码学方面的书，结果发现有些公式与我正在研究的一个与之完全不相干的课题里的公式几乎一样……”。20分钟后，在报告结束时他说：“在我的PC机上恢复这个密钥大约需要两个小时的时间。”

我们应该清楚，对AES进行这样的批评是很不公平的，因为我们目前还没有对AES的攻击方法，而在将来任何密码系统都可能被攻击，当然包括AES。但是，AES简单的代数结构引发了一类完全不同的攻击方法，密码学家们还缺少这方面的知识，在我所接触的密码学家中还没有人了解这些公式。这个世界上也许有人能处理这些公式，但他们大概都不了解密码学。我们可以乐观地估计这种攻击存在的可能性为10%，那么就有10%的机会得到一个实用的攻击算法，也就是说有1%的可能性找到一个针对AES的攻击。而通过使用具有更复杂代数结构的传统密码系统，我们就可以避免这样的危险。

所有的人最终都会使用AES，因为它是美国的政府标准。尽管这个具有清晰代数结构的设计使我们感到不安，但我们仍建议人们使用AES，因为它是标准，这样可以避免一些争论和问题。即使AES被攻破了，也没有人因为你选择了标准密码算法而责备你。

4.5.3 Serpent

Serpent是AES最后5个候选算法之一[2]。毫无疑问，它是所有提交的算法中最保守的一个。Serpent在很多方面与AES相反，AES强调简练和效率，而Serpent更注重安全性。Serpent

共有32轮，我们所知道的最有效的攻击可以攻击其中的10轮[6]。Serpent的缺点是速度较慢，只有AES的1/3，而且算法的有效实现也比较难，因为它的S盒必须转化为适合于底层CPU的布尔函数。

Serpent在某些方面与AES有类似的结构，在每一轮中包括的操作有：与128比特的轮密钥的异或运算、应用线性混合函数以及并行地应用32个4比特的S盒。在同一轮中，这32个S盒完全相同，共有8个不同的S盒轮流使用。

由于Serpent有32轮，在每一轮中要并行地查32个S盒表，共要进行1024次查找，因此速度会很慢。但有一个技巧使得Serpent可以很好地用软件实现，这个技巧就是将S盒表示为布尔公式，即将S盒的4位输出表示为4位输入的布尔函数，这样CPU就可以用AND、OR、XOR指令直接计算S盒的布尔公式。虽然每个S盒的输入可能不同，但寄存器里每个对应位置上的运算是一样的，因此，32位的CPU可以并行地计算32个S盒。这样的实现称为"bitslice"（位片）实现，这是为Serpent特殊设计的实现方法。Serpent的混合运算用这种方法也相对比较容易计算。

如果Serpent与Rijndael（即现在的AES）一样快的话，几乎可以肯定它会被选定为AES。但加密速度的快慢是相对的，实际上Serpent的速度与DES一样，与3DES相比则要快得多，只有和AES的其他几个进入决赛的算法相比，它才显得慢。

4.5.4 Twofish

Twofish也是最后入围决赛圈的5个算法之一。它可以看做是AES和Serpent的折中，其速度几乎与AES一样，而且有较大的安全余地。更重要的是，Twofish没有一个我们所知道的简单的代数表示。Twofish总共有16轮，我们所知道的最强的攻击只能攻击8轮。由于Twofish需要进行大量的关于密钥的预计算，因此，Twofish最大的缺点就是改变加密密钥很不方便。

与DES一样，Twofish也使用Feistel结构，见图4.3[①]。Twofish将128比特的明文分为4个32比特，其后几乎所有的操作都在32比特上进行。轮函数由函数g的两个副本、一个称为PHT的函数F以及一个密钥加法构成，它的输出与右边的那一半做异或运算。图中带有"<<<"或">>>"的盒子表示指定位置的32比特值的循环，而带有"+"的小框代表整数加法。

函数g由4个S盒紧接一个线性混合函数组成，这里的混合函数与AES中的类似，但S盒有所不同。Twofish的S盒与密钥有关，因此Twofish的实现算法要进行预处理，计算S盒并将它保存在内存中。我们采用这种S盒依赖于密钥的设计就是为了增加攻击者对算法分析攻击的难度。

PHT函数采用32位的加法运算，将函数g的两个输出结果混合。最后轮函数F将PHT函数的输出与密钥的相关信息做加法。

① 我们是Twofish的两个设计者，该图直接从我们关于Twofish的书[86]中得到，所以该图比其他图都大而且详细[86]。

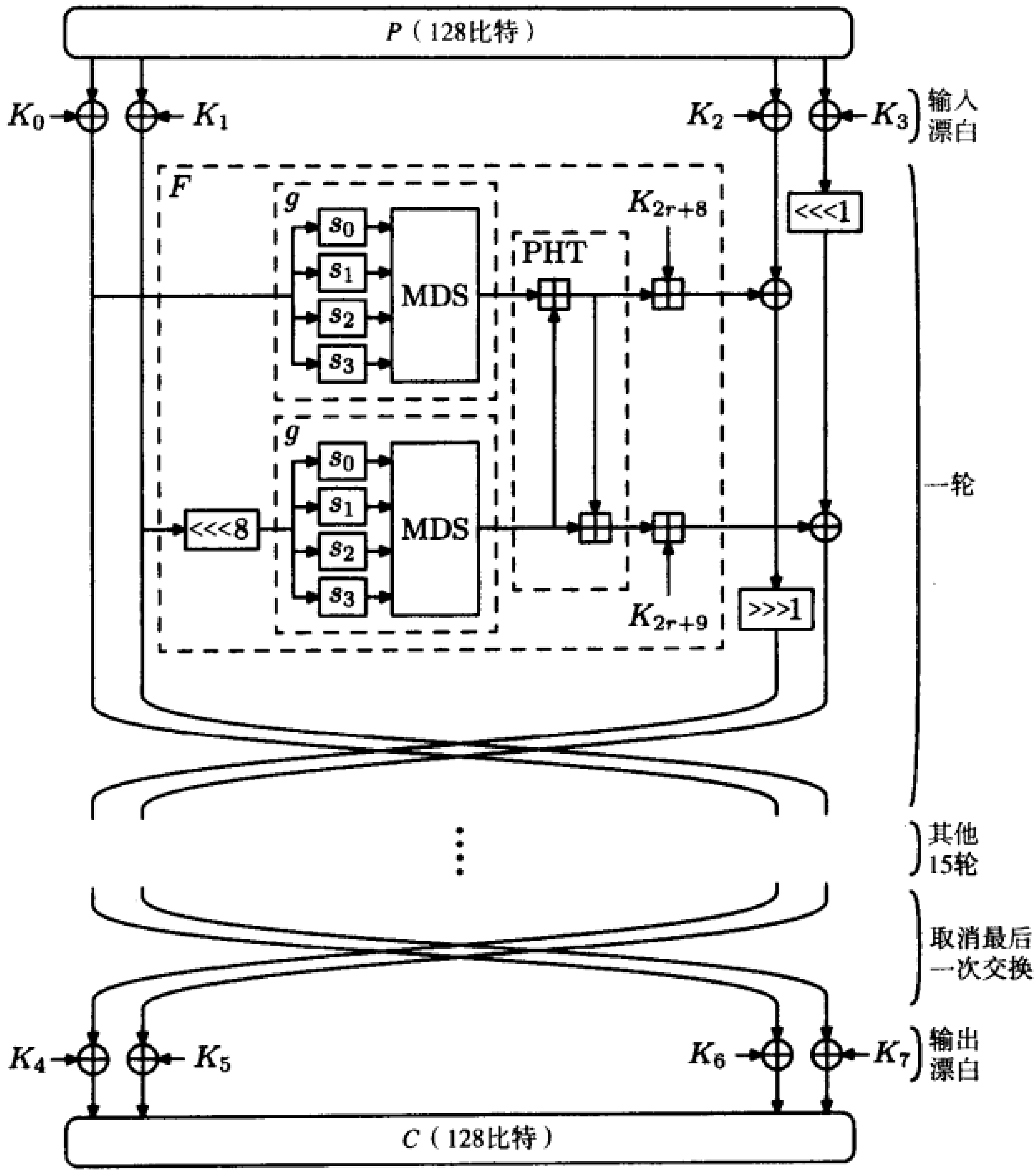


图 4.3　Twofish 结构

引自 Wiley 公司 Niels 的著作（经授权）

Twofish在开始和最后都要与附加的密钥进行异或运算，这称为“漂白”，这样做的开销很小，但却可以使系统抵御多种类型的攻击。

与其他密码算法一样，Twofish也有密钥方案，用于生成轮密钥和“漂白”所用的加法密钥。

应该清楚，我们作为Twofish的设计者，仍然喜欢自己的设计，因此我们的观点将有失偏颇。尽管我们想尽力克服，但也难免有偏见。AES 和 Serpent 设计满足了设计者认为最重要的要求，我们认为Twofish设计满足的一些要求应该成为对一般分组密码的要求。例如，我们在Twofish 里有意增加了两个 1 比特的循环，它能带来两方面的副作用：一方面使得加密算法和解密算法不同，从而增加了实现的代价，另一方面使软件实现的速度降低了大约 5%。不用这些循环将会出现良好的字节排列（byte-aligned）结构，我们增加这些循环的惟一原因就是为了

破坏这种字节排列结构。AES没有破坏它的字节排列结构，我们认为AES也许过于简练了，但AES的设计者也许会认为Twofish的结构不好而且不够精致。这是观点的不同，我们肯定会十分尊重别人的观点，但书中的建议都源于我们自己的观点。

4.5.5 AES的其他候选算法

我们已经详细地介绍了AES的5个候选算法中的3个，另外两个是RC6[78]和MARS[13]。5个候选算法比较起来，我们不太喜欢RC6和MARS，因此不打算详细地讨论。

RC6是一个有趣的设计，共有20轮，其中使用了32比特的乘法。在竞争AES期间，最强的攻击可以攻破17轮的RC6。

MARS的设计具有不一致的结构，使用了大量不同的操作，因此比入围决赛的其他算法的实现代价更高。除此以外，MARS还有两个错误。一个是在生成S盒的软件编码中有隐错，导致得到的S盒不能满足MARS研究小组为S盒确定的标准。另一个错误更严重，设计者给出的MARS抵抗线性密码分析（一种特殊的攻击方法）的证明中有一个严重的失误，对MARS抵御线性密码分析的性能至今没有一个理想的分析结果。线性攻击是密码学界熟知的、攻击力很强的攻击方法，我们希望任何严谨的分组密码都能有这方面的分析。

RC6和MARS也许是两个很好的分组密码，但我们认为AES、Serpent与Twofish这3个更好，应该从这3个中选择使用。

4.5.6 求解方程攻击

当我们写这本书时，一种新的攻击类型正在发展，引起了密码学界的关注。它的基本思想是将分组密码的加密写为有限域上的一组线性和二次方程，然后用新近发展的技术（称为XL、FXL和XSL）对这些方程进行求解。

Nicolas Courtois和Josef Pieprzyk在2002年声称可以用这些技术攻击Serpent和AES[17]，这在整个密码学界引起了震动，他们声称的对AES的攻击受到了极大的关注。我们感到特别震惊的是对32轮完整的Serpent的攻击，因为我们认为Serpent是所有AES候选算法中最保守的一个。

在这些结果公布以后，有人认为Courtois和Pieprzyk的攻击仅仅是理论上的，实际上不会成功。XSL方程的求解算法很复杂，而且对所要求解的方程的形式也十分敏感，对要攻破AES和Serpent所需的工作量的估计也有很大程度的试探性，到目前为止，估计的工作量仍大于2^{128}，所以XSL攻击对我们总是采用128比特的设计强度所设计的系统没有影响。因此，只要XSL攻击没有大的改进，就不会对我们的任何实际系统构成威胁，但作为一种新的攻击技术，我们估计它在将来会有所改进。

据我们所知，现在还没有人实现这些技术。我们希望能看到使用这些技术对减少轮数的AES和Serpent进行攻击的计算机程序。由真实密码系统得到的现实经历，我们可以推断出关

于攻击这几个密码的XSL算法的性能。由于没有这样的经历，我们确实不知道求解方程攻击是否有效。

我们不知道未来会如何，但要问今后5年，我们也许会知道一些。求解方程攻击完全是新的一类攻击，目前还没有为抵御它而设计的密码。如果XSL式的技术确实有效，那么我们现在使用的所有密码系统都可以被攻破。如果有一个密码系统能够存活，那完全是侥幸。另一方面，按照我们的观点，XSL式的技术实际上根本没有用，或者仅对少数高度结构化的设计有用。

对Twofish又如何呢？现在还没有人试图将这些技术应用于Twofish，因为将这些技术用于攻击Twofish远比攻击Serpent或AES要复杂。因此，我们还不知道这些技术在攻击Twofish时是否有用，但如果XSL被证明可以有效地攻击AES和Serpent，那么无疑它也可以用于攻击Twofish。

4.5.7 应该选择哪一个分组密码

切记，我们是有偏见的，因为我们是Twofish设计小组的成员，而且我们也花费了大量的时间去攻击AES的其他候选算法，这进一步影响了我们的观点。

我们不知道XSL式的攻击如何影响分组密码的安全性，我们几乎没有什么资料，而为数不多的已发表的结果还是有争议的，因此，XSL式的攻击使得选择一个分组密码变成了一个很棘手的任务。由于还没有设计出能抵抗这类攻击的分组密码，现有的分组密码都是脆弱的，我们认为密码系统的选择不应该受XSL攻击的影响。现在，还没有足够的信息能够判断一个分组密码比另外一个更容易受到XSL式的攻击。

对你来说，AES是一个安全的选择，这是美国政府认可的官方标准，而且其他人也在使用它。我们认为，虽然这个选择不能保证数据的绝对安全，但如果有一个针对AES的攻击获得成功，那也不是你的错。人们常说“没有人因为购买IBM而被解雇”，同样，没有人因为你选择了AES而解雇你。只要不因为AES被攻破而遭受经济损失或失眠之苦，那就选择AES。

AES的另外一个优点是易于实现和使用。所有的密码学书籍都会论述它，所有的客户因为它是标准而喜欢它。从这个意义上看，选择AES是不会错的。

如果你特别关心数据的安全性而不在乎速度，可以选择Serpent。在确定AES算法的过程中，所有密码学家都认为Serpent是所有提交的算法当中最安全的一个。

这里没有为Twofish提供很多的机会，如果想拥有AES的速度而且避免以上列举的安全性缺陷的话，应该选择Twofish。当然，你要为此承担责任，一旦Twofish被攻破，你将为你的选择受到责备。

也许对有些环境来说，3DES仍然是最好的选择。如果必须向下兼容，或者受系统其他部分的限制只能采用64比特的分组长度，那么3DES仍然是最好的选择，但要切记3DES不满足我们的安全性标准，而且使用64比特的分组长度时应特别小心。

4.5.8 密钥应为多大

分组密码可供选择的3个主要算法（AES、Serpent和Twofish）都有128比特、192比特或256比特3种长度的密钥。对几乎所有的应用来说，128比特的安全水平已足够了，但我们一点也不喜欢采用128比特的密钥。

除了碰撞攻击外，128比特的密钥已很大了。我们多次发现密码系统被生日攻击或中间相遇攻击攻破。这些攻击方法确实存在，但设计者有时恰恰忽略了它们。有时设计者认为它们是安全的，但有人却找到了一个非常巧妙的使用这些攻击的新方法。大多数分组密码模式都存在某种形式的中间相遇攻击，对此我们有丰富的经验，我们的设计准则是：

设计准则 3： 要求有 n 比特的安全水平，所有的密码值至少应为 $2n$ 比特。

这个准则的实现代价相对较低，而且它还可以使任何形式的碰撞攻击无效。在现实生活中，很难严格遵循这个准则。为获得128比特的安全水平，我们确实想采用分组长度为256比特的分组密码，但所有普通分组密码的分组长度都是128比特。这个问题实际上更严重，对分组密码已有相当多的碰撞攻击。

尽管如此，我们至少可以使用AES候选算法支持的长密钥，即使用256比特的密钥。

很遗憾，AES算法（Rijndael）用256比特的密钥比用128比特的密钥要慢①。当使用AES时，也许迫于对性能的要求而使用较小的密钥。我们不是说128比特的密钥就不安全，而是说使用它必须格外小心。用128比特的密钥使系统获得128比特的安全性是可能的，为此，不能简单地使用标准的分组密码模式，必须为它配备额外的密钥信息以抵抗碰撞攻击。这使得系统很复杂，从而导致存在一些弱点。因此我们建议使用256比特的密钥。

我们主张设计强度为128比特的系统时使用256比特的密钥。换句话说，这些系统设计可以抵御那些能完成 2^{128} 次运算的攻击方法的攻击。要记住，应采用设计强度（128比特）而不是256比特的密钥长度来规划系统的其他部分。

使用256比特的密钥获取128比特的设计强度还有一个更复杂的因素。前面提及的XSL式攻击用于较大密钥的AES和Serpent时比穷举搜索更有效，对Serpent的攻击的操作比 2^{128} 步要多。如果我们让Serpent的设计强度为128比特，即使XSL可以像声称的那样攻击256比特的Serpent，那它也不是256比特的Serpent的攻击。如果设计强度为128比特，那么攻击就必须比在 2^{128} 元素上的穷举搜索更有效，而XSL攻击对Serpent和AES都需要比这更多的操作。

① Serpent对任意大小的密钥速度都一样。Twofish初始化大密钥的速度较慢，但用软件实现时加密速度与密钥的大小无关。

第5章 分组密码模式

分组密码只能加密固定大小的明文块。如果要加密的明文的大小不是分组的大小，则需要使用分组密码模式。

在开始阅读本章之前，我们要提醒读者：加密模式可防止窃听者获得通信信息，但不能提供认证，即攻击者仍然可以按照自己的愿望修改消息。许多人对此感到惊奇，其实这很常见。解密函数仅仅对加密的数据解密，如果密文已被更改，解密可能产生一个没有意义的结果，但它也会通过解密得到（被修改或没有意义的）明文。不要以为一个没有意义的消息不会带来恶果，其实它常常会导致不幸事件的发生，这与系统的其他部分有关。

在几乎所有的情况下，修改消息带来的危害远比泄露明文的危害更大，因此，加密和认证总是结合在一起的。本章讨论的加密模型应该与一个独立的认证函数相结合，关于认证函数将在第7章叙述。

5.1 填充

分组密码模式是将任意长的明文P加密产生密文C的一种方法，大部分模式都要求明文的长度刚好是分组大小的整数倍，因此需要对明文P进行填充。明文填充的方法有很多，基本原则是填充必须是可逆的，即能够从填充后的消息惟一确定原始消息。

我们看一个非常简单的填充方法，即在消息的最后添加足够多的“0”以使长度满足要求。这个方法不好，因为它不可逆，如明文p与$p \| 0$有相同的填充形式（$\|$表示连接）。

在本书中，我们总是假定明文由整数字节组成。有些密码学的基本构件的大小被指定带有零头，其中最后一个字节没有完全使用。这不但没什么用，反倒会成为一个障碍。许多的具体实现也不允许使用带零头方式，因此我们所说的大小是按字节说的。

最好能有一种填充规则，使得在明文长度合适的情况下不再增加明文的长度，但这是不可能的。对任意的可逆填充，至少可以找到一个消息，它的长度恰好合适，但必须由于填充而加长。实际上，所有的填充规则至少使明文长度增加一个字节。

那么，如何填充明文呢？设P是明文，$l(P)$表示明文P的字节数，b是分组密码分组长度的字节数。我们建议使用以下两个简单填充方案中的一个，其中任意一个都能满足要求。填充没有密码学意义，任意一个可逆的填充都可以采用，我们给出的是两个最简单方法。

1. 首先添加值为128的一个字节，然后添加一些0字节使得总长度为b的倍数，这里0字节的数量在0与$b-1$之间。

2. 首先确定需要填充的字节数 n，即选择 $1 \leqslant n \leqslant b$，使得 $n + l(P)$ 是 b 的倍数，然后填充 n 个值为 n 的字节。

在完成填充从而使明文的长度为分组长度的整数倍后，我们将填充后的明文分成许多块，这样明文 P 就变成块序列 $P_1, \cdots, P_k$，其中块的个数为 $k = \lceil (l(P) + 1)/b \rceil$，其中，$\lceil \cdots \rceil$ 表示不小于下一个整数的最小整数，即 ceiling 函数。在本章我们一般假定明文 P 由明文块 $P_1, \cdots, P_k$ 组成。

在用下面将要介绍的某个分组密码模式完成解密后，这些填充必须去掉，同时要检查填充的正确性，必须验证填充的每个字节都有正确的值，对错误的填充要当做认证失败处理。

5.2 ECB

最简单的加密长明文的方法称为电子密码本模式（ECB, electronic codebook），定义为

$$C_i = E(K, P_i) \qquad i = 1, \cdots, k$$

即用加密算法分别加密每一个明文块。当然，事情不会这么简单，否则我们就没有必要用一章的篇幅论述分组密码模式了。ECB 有严重的缺陷，这里之所以列举它是为了告诫读者无论如何都不要使用它。

用 ECB 会有什么问题呢？在 ECB 模式中，如果两个明文块相同，则它们对应的密文块也一定相同，攻击者也知道这一事实。ECB 的这个密文依赖于消息结构的特性将会给攻击者泄露大量的信息。

在很多情况下，明文中会有大的文本块重复出现。例如在本章的很多地方都提到了“密文块”（“ciphertext block”），如果密文块两次都出现在块边界，那么将重复相同的明文块。用统一的字符编码标准进行编码，所有其他字节都为 0，这大大增加了重复块出现的机会。总之，ECB 的这个缺陷使它脆弱得无法使用。

5.3 CBC

密码分组链接模式（CBC, cipher block chaining）是一种广泛使用的分组密码模式，它将明文块与前一个密文块进行异或运算，然后再加密得到对应的密文块，从而克服了 ECB 的缺陷。CBC 的标准形式定义为：

$$C_i = E(K, P_i \oplus C_{i-1}) \qquad i = 1, \cdots, k$$

这相当于用前一个密文块对当前的明文块进行了“随机化”，因此相同的明文块将对应不同的密文块，从而大大减少了信息的泄露。

5.3.1 固定初始向量（IV）

使用 CBC 模式还有一个问题，即要确定 C_0 的值。这个值称为初始向量（initialization vector）或 IV。不应使用固定的 IV，否则对每一个消息的第一个块会出现 ECB 的问题。如果两个不同

的消息有相同的开头，即最前面的块相同，那么加密将从相同的密文块开始。在现实生活中，大量的消息都有完全相同或相似的开头，而我们不想让攻击者了解这些。

5.3.2 计数 IV

我们知道的一个可供选择的方法是使用计数 IV。对第一个消息，让 IV = 0；对第二个消息，让 IV = 1。然而，这也不是一个非常好的主意。正如前面提到的，现实生活中很多消息的开头都是类似的，如果消息的第一个块的差别很小，计数 IV 有可能去掉这个差别，从而再次产生同样的密文块。例如，0 与 1 仅在一个位置上不同，如果前两个消息开头的明文块恰好只在这个位置上不同（这种事情的发生可能要比你想像的多），那么两个消息的第一个密文块相同。攻击者可以迅速地发现两个明文之间的差别，这是安全的加密方案所不允许的。

5.3.3 随机 IV

电子密码本模式（ECB）与采用固定 IV 以及计数 IV 的密码分组链接模式（CBC）存在的问题都源于明文消息极强的非随机性，明文通常有固定的开头，或有非常清楚的结构。在CBC中，我们用密文块将明文块“随机化”，但对第一个明文块，则必须使用初始化向量 IV 将其随机化，因此应该采用随机的 IV。

随机选择 IV 的问题是随机 IV 必须让接收者知道，如此才能正确地解密。标准的解决方案是选择随机 IV 并作为第一个密文块放在消息对应的密文之前发送给接收者，这样，加密过程可以描述为：

$$C_0 := \text{随机块值}$$
$$C_i := E(K, P_i \oplus C_{i-1}) \qquad i = 1, \cdots, k$$

这时明文 $P_1, \cdots, P_k$ 就被加密为 $C_0, \cdots, C_k$。注意，密文是以 C_0（而不是 C_1）开始的，可见密文要比明文长。对应的解密过程为：

$$P_i := D(K, C_i) \oplus C_{i-1} \qquad i = 1, \cdots, k$$

使用随机 IV 并不是一个完美的解决方案，它也有两方面的缺陷。首先，加密算法必须访问一个随机资源，而得到一个理想的随机发生器是一件困难的事，因此最好避免这样做。其次，密文比明文大一个分组长度，对短消息来说，密文的这种增大不容忽视。

5.3.4 瞬时 IV（Nonce-Generated IV）

瞬时 IV 是 IV 问题的一个最好的解决方案，由两步组成。首先，为每一个用该密钥加密的消息分配一个惟一的数，称为瞬时值（nonce，由 number 的第一个字母与单词 once 合成），然后对瞬时值加密得到 CBC 必需的 IV。瞬时值的关键是它的惟一性，对同一个密钥，相同的瞬时值不能使用两次。典型的瞬时值是消息的某种编号，或由编号与其他信息相结合而构成。消息编号用于保证消息的正确顺序、检测消息重放等，在很多系统中都可使用。瞬时值本身并

不需要保密，但一个瞬时值只能使用一次。接收者与每个消息一起同时收到一个瞬时值，对接收者来说重要的是收到的瞬时值没有重复。

在通常情况下，发送者对消息连续地编号并发送这个消息编号。发送者发送消息的过程如下：

1. 为这个消息分配一个消息编号。消息的编号由一个从0开始的计数器提供，而计数器在任何时候都不能返回到0，否则惟一性将遭到破坏。
2. 用消息编号构造惟一的瞬时值，它的大小为分组密码的分组长度。对给定的密钥，瞬时值不仅在这个计算机中，而且在整个系统都是惟一的。例如，同一个密钥用于加密双向的通信，这时瞬时值应该由消息编号与标志这个消息的发送方向组成。
3. 对瞬时值加密得到IV。
4. 用这个 IV 在 CBC 模式下对消息加密。
5. 在密文上附加足够多的信息，以使接收者能恢复这个瞬时值。一般的方法是在密文的前边附加上消息编号。IV 本身（公式中的 C_0）不必发送。
6. 确保接收者收到的所有消息编号都是惟一的。为此，要求接收者拒绝那些小于或等于以前接收到的消息编号的编号。

当使用瞬时IV时，必须包含在消息中的附加信息比在随机IV时少得多。对大多数系统来说，32~48 比特的消息计数器已经足够大了，而随机 IV 需要 128 比特的 IV。另一方面，大部分通信系统总要有消息计数器，因此以这种方式生成 IV 并没有增加开销。

5.4 OFB

到目前为止，所有的模式都接受消息，然后以某种方式用分组密码对消息块进行加密。输出反馈模式（OFB, output feedback）与此不同，它不将消息作为加密函数的输入进行加密。输出反馈模式首先用分组密码生成一个伪随机字节流（称为密钥流），然后将其与明文进行异或运算得到密文。这种用生成随机密钥流进行加密的方法称为流密码。一些人会认为流密码在某些方面不好，其实不是这样。流密码非常有用，而且只要在使用时加以注意，它的性能就会稳定。重复使用瞬时值是滥用流密码的重要形式，这种不正确的方式很容易使系统变得非常不安全。像 CBC 这样的模式更健壮，即在被错误使用的时候仍然能正常工作。相比之下，流密码的优点比它的缺点更突出。

OFB 可以定义为：

$$K_0 := \text{IV}, K_i := E(K, K_{i-1}), C_i := P_i \oplus K_i \qquad i = 1, \cdots, k$$

OFB 将 IV 作为 K_0，重复进行加密生成密钥流 $K_1, \cdots, K_k$，然后将明文与密钥流进行异或运算产生密文。

OFB 中 IV 必须是随机的，如同在 CBC 中一样（见 5.3.3 节），IV 可以随机地选取并同密文一起发送出去，也可以由一个瞬时值生成（见 5.3.4 节）。

OFB的一个优点就是它的加密运算和解密运算完全一样，实现起来很省事，仅需要实现分组密码的加密函数，无需再实现解密函数。

OFB的另一个优点是不需要对明文进行填充。将密钥流当做一个字节序列，加密时可以只使用和明文一样多的字节。换句话说，如果最后一个明文块的一部分是有效的，那么就可以只发送那些和明文对应的密文字节，多余的可以丢掉。加密前不对明文进行填充减少了系统开销，这对短消息来说特别重要。

由于使用流密码，OFB有一个潜在的危险。一旦有两个不同的消息使用了同一个IV，那么这两个消息就被相同的密钥流加密。这是非常糟糕的事情。假定P与P'是两个不同消息的明文，它们被相同的密钥流分别加密为密文C与C'，现在攻击者可以计算$C_i \oplus C'_i = P_i \oplus K_i \oplus P'_i \oplus K_i = P_i \oplus P'_i$。也就是说，攻击者可以算出两个明文之间的不同。如果攻击者已经知道了其中一个明文（这在现实生活中常常可能发生），他可以很容易地求出另外一个明文。有些著名的攻击甚至可以从未知明文的差别来恢复这两个明文[45]。

OFB的另一个问题是：如果一个密钥块重复出现，那么密钥块序列就会出现循环。对一个较大的消息，密钥块序列有可能陷入一个循环中。另外，如果一个消息的IV与另外一个消息所对应的密钥块序列中的某一个密钥块相同，则两个消息的一部分就使用了相同的密钥块序列。在这两种情形下，都出现了用相同的密钥流对不同的消息进行加密的现象，这是一种非常不安全的加密方法。

要确保系统能加密很多的数据而且不会发生碰撞。碰撞主要发生在密钥流块和初始点之间，所以希望在发生碰撞之前至少能加密2^{64}个数据块。如果限制每个密钥所能加密的数据量，当然可以降低密钥块重复的可能性，但遗憾的是危险依然存在，假如不幸出现了碰撞，整个信息的机密性也就丧失了。

5.5 CTR

我们所偏爱的分组密码加密模式是计数模式（counter mode），简称为CTR。这种模式已使用多年，但却没有被作为DES的标准模式[69]，因此被一般的教科书所忽视。最近，美国国家标准和技术研究所（NIST）将CTR进行了标准化[27]。与OFB一样，CTR模式也是一种流密码模式，定义为：

$$K_i := E(K, \text{Nonce} \parallel i), C_i := P_i \oplus K_i \qquad i = 1, \cdots, k$$

同所有的流密码一样，加密时必须使用某种形式的惟一瞬时值，而大多数系统都用消息编号与某些附加信息来生成瞬时值，以保证它的惟一性。

CTR生成密钥流的方法很简单。首先将瞬时值与计数值连接起来，然后对此进行加密产生密钥流的一个密钥块。显然，这里要求瞬时值与计数值的连接不能超过单个块的大小，对现在128比特的分组来说，这几乎不是问题。很明显，由于需要存放计数值i，瞬时值长度必须小于

块的大小。对于 128 比特的分组，典型的比特分配方案是消息编号占 48 比特，瞬时值的附加数据占 16 比特，而其余 64 比特用于计数器 i。这样将限定系统使用一个密钥只能加密 2^{48} 个不同的消息，而每个消息最多能有 2^{68} 个字节。

同 OFB 模式一样，在 CTR 模式中绝对不能重复使用一个密钥和瞬时值的组合，这是经常提及的关于CTR 的缺点。CBC 模式也有同样的问题，只不过 CBC 略微健壮一些，即若两次使用同一个 IV 将泄露关于明文的信息，但所泄露的信息量受到限定。但是，任何泄露都违背了安全性要求。在模块化的设计中，即使只有一个模块有很小的信息泄露，其危害也可能是巨大的。我们不能希望系统的其他部分也限定由少量信息泄露所带来的危害，因此，在使用CBC和CTR 时，必须保证 IV 或瞬时值的惟一性。

除此以外，CTR 模式很容易使用，仅需实现分组密码的加密函数即可，因为它的加密函数与解密函数一样。同时，CTR 模式的密钥流中的任意一个密钥块都可以即时计算，因此可以非常容易获取明文的任意一部分，而且若并行地计算密钥流，则可获取更快的速度。另外，CTR 的安全性与分组密码的安全性之间的关系是平凡的，CTR加密模式的任何一个缺陷都隐含着分组密码的一个选择明文攻击。反之，如果不存在对分组密码的攻击，则也不存在对 CTR 模式的攻击。

5.6　一些较新的模式

我们前面介绍的所有模式都产生在 20 世纪 70 年代到 80 年代早期，最近几年又提出了一些新的模式，其中 OCB 模式可能是最引人注意的一个[83，84]。这个模式实现了认证和加密两个功能，具有极大的吸引力，但我们还是建议不要使用它，理由有两个：

第一，OCB 是一个很新的模式，而在这个领域里，新方案的可靠性总是被置疑的。很多新模式（包括 OCB）都有一个安全性的证明，但是我们不能相信这些证明，因为证明本身就不可信。过去的一些安全性证明后来被证实是错误的，将来也是如此。这样的证明通常都很复杂，认为这些证明本身没有瑕疵有点过于乐观（我们还没有见到任何证明其正确性的尝试）。

安全性的证明并不是证明这个系统是安全的，而是给出了一个安全性的归约（security reduction）。例如，证明的结论可能陈述为如果可以在 X 时间内攻破 OCB，那么就可以在 Y 时间内攻破基本的分组密码。这是个有价值的成果，而且恰巧能适合我们的模块化密码设计。然而，这只证明了分组密码模式的安全性，而不是使用这个模式进行加密的安全性。安全性的证明很令人头疼，有这样的证明肯定比没有好，只可惜这些证明都不能令人相信。

第二，这些模式的专利情况还不是很清楚，不同的团体声称的专利涉及各种各样的模式。在我们编写本书的时候，这些专利还没有发布，每个专利所涉及的确切内容还不清楚。使用其中的任意一个模式都有承担责任的危险，因此要小心。

5.7 模式的选择

我们已经介绍了好几种模式，但其中只有两种模式可以考虑使用，它们是CBC和CTR。我们在前面已经说明ECB不够安全，而OFB尽管不错，但CTR在某些方面比它更好，并且没有短循环问题，因此没有理由丢弃CTR而选择OFB。

那么，在CBC和CTR中应该选择哪一个呢？下面给出它们在几个方面的比较结果：

- **填充**：CBC需要对消息进行填充，而CRT不需要。
- **速度**：这两种模式有相同的计算量，但CTR允许进行任意的并行计算，因此可以获得更高的速度。
- **实现**：CTR仅需要分组密码的加密函数，而CBC则要求同时实现分组密码的加密函数和解密函数。
- **健壮性**：如果重复使用了一个相同的瞬时值，CBC可能会泄露有关原始明文块的一些信息，而CTR将泄露有关整个消息的信息。
- **瞬时值**：CBC可以使用随机IV或一个瞬时值，而CTR要求有惟一的瞬时值，但在实际使用时，CBC模式的加密总是使用一个瞬时值，因此在这方面两个模式相同。

由此可见，除了健壮性外，CTR在几个方面都占优。而在大多数系统中，设计者可以自己设计瞬时值系统，因此很容易克服瞬时值重复使用的问题。以我们的观点，CTR有足够多的优点让你去选择它而不是选择CBC，除非你不能控制加密函数的使用方式。

前面我们曾说系统的每一部分都应该是安全的，不应依赖于系统的其他部分，但是我们这里所推荐的CTR模式的安全性却依赖于系统其他部分为它提供惟一的瞬时值，我们的观点自相矛盾吗？答案既是肯定的，也是否定的。我们期望一个分组密码模块不依赖于整个系统的其他任何部分，但是我们所讨论的加密模式都没有这样的性质，它们在一定程度上都依赖于系统的其他部分，这是因为分组密码模式不是一个完整的加密系统，也不是独立设计的模块。分组密码是一个模块，它可以被另一个合适的分组密码所取代，而分组密码模式仅仅是我们在下一个层次模块（加密-认证-消息模块）中使用的一种方法。这个模块将在第8章中论述。由于它不是一个独立的模块，因此不能希望分组密码模式的安全性独立于系统的其他部分。

加密模式仅仅提供机密性，也就是说，攻击者不能了解你的通信数据，而不是你正在通信这件事本身以及通信的时间、通信量和通信对象这些事实①，而且数据加密也不能阻止攻击者对数据进行修改。

① 对这些事实的分析称为通信量分析，它可以为攻击者提供很多有用的信息。防止通信量分析是可能的，但带宽要求对军方以外的人来说太高了。

5.8 信息的泄露

我们现在揭示有关分组密码模式的一个秘密。任何分组密码模式都存在信息的泄露，没有一个是完美的。这方面的分析在文献中很难看到，因此我们在此予以简述。

我们假定已有一个完美的分组密码，但即使采用这个分组密码，由加密模式加密产生的密文仍然会泄露有关明文的一些信息。这与明文块以及密文块的相等与否有关。

我们首先来分析 ECB。如果两个明文块相等，即 $P_i = P_j$，那么对应的密文块也相等，即 $C_i = C_j$。对随机的明文来说，这样的事情很少发生，但大多数的明文不是随机的，而且有很强的结构。明文块之间的相等很常见，而密文块之间的相等关系会泄露明文的这种结构，这就是舍弃 ECB 的原因。

CBC 又如何呢？由于每一个明文块先与前一个密文块进行异或运算后再加密，从而使得相等的明文块所对应的密文块不一定相等。设想所有的密文块是一些随机值，它们是由分组密码给定输入产生的随机输出。如果我们有两个相等的密文块，会怎么样呢？设 $C_i = C_j$，即

$$E(K, P_i \oplus C_{i-1}) = E(K, P_j \oplus C_{j-1})$$

两边解密得

$$P_i \oplus C_{i-1} = P_j \oplus C_{j-1}$$

从而有

$$P_i \oplus P_j = C_{i-1} \oplus C_{j-1}$$

最后一个公式说明两个明文块的差别等于对应密文块的异或，而攻击者知道密文块的异或。这一定是在任何一个完美的加密系统中都不该发生的事情。另外，如果明文的冗余度很大（如英文文本文件），则可能包含足够的信息可由 $P_i \oplus P_j = C_{i-1} \oplus C_{j-1}$ 恢复这两个明文块。

当两个密文块不相等时，情况也类似。由 $C_i \neq C_j$ 可得 $P_i \oplus P_j \neq C_{i-1} \oplus C_{j-1}$，于是互不相等的一对密文块隐含了关于明文块的一个不等式。

CTR 具有类似的性质。在 CTR 加密模式中，由于密钥块 K_i 是由瞬时值与计数值加密得到的，它们一定互不相同。对任意给定的两个密文块 C_i 与 C_j，必有 $P_i \oplus P_j \neq C_i \oplus C_j$，否则就应有 $K_i = K_j$，这是不可能的。换句话说，CTR 模式同样隐含着关于明文块的一个不等式。

由于在 CTR 中任意两个密钥块都不相同，所以由两个密文块相同或两个明文块相同并不能获取更多信息，因此 CTR 没有碰撞问题，这也是 CTR 有别于理想的流密码的惟一特征。

OFB 比 CBC 和 CTR 都更糟糕。对于 OFB，只要没有碰撞发生，它与 CTR 泄露同样多的信息；如果出现碰撞，即有两个密钥块相同，那么后续的密钥块也发生碰撞。这对安全性构成了灾难性的威胁，也是 CTR 比 OFB 更受欢迎的理由。

5.8.1 碰撞的可能性

现在要问，两个密文块相等的机会有多少？假设我们总共加密了 M 个明文块（设块的长度为 n），至于这 M 个明文块属于多少个消息无关紧要。M 个块中任意两个组成一对，共可以组成 $M(M-1)/2$ 个不同的对，而两个块相等的可能性为 2^{-n}，因此，密文块相等的数量的期望值是 $e = M(M-1)/2^{n+1}$。当 $M \approx 2^{n/2}$ 时，$e = M(M-1)/2^{n+1} \approx 1$。这就是说大约加密 $2^{n/2}$ 个明文块就可以期望得到两个相等的密文块[①]。当 $n = 128$ 比特时，大约加密 2^{64} 个明文块就可以期望得到两个相等的密文块。这就是已在3.6.6节说明的生日悖论（birthday paradox）。现在来看，2^{64} 个块是一个庞大的数字，但不要忘记我们设计的系统应有30年的使用寿命，在将来也许人们会处理接近 2^{64} 个块的数据。

另一方面，即使较小的数据量也会有危险。假如我们处理 2^{40} 个块（大约16 TB的数据量），那么就有 2^{-48} 次机会得到一对相等的密文块。这确实是一个很小的概率，但对攻击者来说，他可以取定一个特殊的密钥，收集 2^{40} 个块并检测重复的块。这样做成功的机会当然很小，但他可以对大约 2^{48} 个密钥重复这个工作，于是攻击者发现一个碰撞的计算量为 $2^{40} \cdot 2^{48} = 2^{88}$，这远小于我们的128比特的设计强度。

现在来分析CBC和CTR。在CTR中只能得到每个明文块的不等性，而对CBC来说，由密文块的不等性或相等性可以对应地得到关于明文块的不等性或相等性。显然，相等性要比不等性泄露给攻击者的信息更多，因此CTR比CBC泄露的信息少。

5.8.2 如何处理信息泄露

我们如何达到128比特的安全目标呢？从根本上讲，这是不可能的。没有什么简单的方法能用分组大小为128比特的分组密码来获得128比特的安全水平。为了达到这样的安全水平，我们需要256比特的分组密码，但目前相关的研究还不够，因此陷入了僵局。我们目前所能做的就是尽量接近设计的安全水平，减轻由此带来的损害。

CTR泄露的信息很少。假设我们加密了一个 2^{64} 个块长的数据，得到了密文 C。对任意一个可能的明文 P（长为 2^{64} 个块），攻击者可以求出用于将明文 P 加密为密文 C 的密钥流，而且密钥流包含一个碰撞的机会大约为50%。我们知道CTR不会有碰撞发生，一旦有碰撞发生，这个特殊的明文就可以被排除。因此，在所有可能的明文 P 中，攻击者可以排除大约一半的明文，这对应于1比特的信息泄露。如果我们仅加密 2^{48} 个块，那么攻击者大约可以排除所有可能明文的 2^{-32}，攻击者几乎没有得到什么信息。在实际应用的环境下，这样少的泄露没有利用价值。因此，尽管CTR不完美，但我们可以通过限制一个密钥不要加密太多的信息来减少它的危害。限制加密不超过 2^{60} 个分组块也许是合理的，这也就是说加密 2^{64} 个字节的信息仅泄露少于1比特的信息量。

① 在期望第一对相等的密文块出现之前，可以加密的块数的更准确的估计为 $\sqrt{\pi 2^{n-1}} = 2^{n/2}\sqrt{\pi/2}$。这个结果的理论分析很难，而且我们这里并不需要这么精确的估计。

当使用CBC时应增加更严格的限制，因为一旦发生了碰撞，将会泄露128比特明文信息。降低碰撞发生的概率是保证安全性的一个很好的策略。我们建议，用CBC加密的数据量应在2^{32}个块左右，这使得泄露128比特信息量的概率为2^{-64}。当然，这离我们期望的安全水平还很远，但对大多数应用来说也许足够了。

注意，这些限制是对可以用一个密钥进行加密的数据总量的限制，这些数据可以来自一个很长的消息，也可以由很多的短消息组成。

目前的情形不能令人满意，但这确实是我们所面对的情形。对此，我们所能做的就是选择CTR或CBC，并且对用一个密钥所处理的数据量加以限制。当一个密钥将要到期时，建立一个新的密钥是很容易的事。我们将在后面讨论密钥协商协议。假如你已经用密钥协商协议建立了加密密钥，那么更新密钥不是很困难，但是比较复杂。这些方法也许不受欢迎，但目前这是最好的实用方法。

5.8.3　关于我们的数学

具有数学背景的读者也许会对我们使用概率的简单方法感到惊讶，因为我们没有验证有关的独立性。当然，从纯数学的角度来看，他们的看法是有道理的。但是，正如物理学家一样，密码学家用他们发现的更为有用的方式使用数学。密码学家们竭尽全力地去破坏所有的模式，因为模式会导致攻击。经验证明，这种处理概率的方式能得到相当精确的结果。数学家可以对此进行更深入细致的研究，获得精确结论，但我们还是喜欢用这种近似而简明的方法。

PDG

第 6 章 Hash 函数

Hash 函数是所有密码学基元中用途最多的一个，它可以用于加密、认证，甚至可以构造简单的数字签名方案①。

Hash 函数的输入是任意长度的比特或字节串，而输出是固定长度的对应串。Hash 函数的一个典型的应用是数字签名。对给定消息 m，我们当然可以直接对这个消息本身进行签名，然而，大多数数字签名在用于公钥运算方面的开销相当大，直接对消息本身签名很不经济，因此可以先对消息 m 应用 Hash 函数 h，然后对 $h(m)$进行签名。Hash 函数的输出一般在 128 比特到 512 比特之间，与很长的消息 m 相比，对 $h(m)$签名比对消息 m 直接签名要快得多。为保证这样构造的签名的安全，必须要求 Hash 函数 h 满足以下条件：构造两个消息 m_1 和 m_2，使得 $h(m_1) = h(m_2)$是不可能的。下面我们详细讨论 Hash 函数的安全性。

Hash 函数有时又称为消息摘要函数，其值叫做摘要或指纹。除生成消息摘要外，Hash 函数还有很多其他的应用，我们喜欢用更通用的名字——Hash 函数。在访问一种很多算法中使用的称为 Hash 表的数据结构时，Hash 函数这个名字又表示一种映射函数。这种 Hash 函数与密码学中的 Hash 函数有一些相似的性质，但二者有很大的不同，不要因为有相同的名字而将二者混淆。我们在密码学中使用的Hash函数有特殊的安全性要求，而Hash表的映射函数（Hash 函数）并不需要这些性质。本书中的 Hash 函数是指密码学中的 Hash 函数。

Hash 在密码学中有很多的应用，它在密码系统的各个不同部分之间建立紧密的联系，只要拥有一个大小可变的值，总可以用Hash 函数将其映射为一个具有固定长度的值。Hash 函数可以作为密码学的伪随机发生器从一个共享秘钥生成几个密钥。它所具有的单向性可以起到隔离系统不同部分的作用，以保证即使攻击者获悉了其中一个密钥，也不能获得其他值。

尽管几乎所有的系统都要使用Hash 函数，但与对分组密码的研究相比，我们对它的了解却很少。人们在 Hash 函数方面所做的研究工作甚少，也没有很多方案供我们选择。

6.1 Hash 函数的安全性

Hash 函数将任意长度的消息 m 映射为固定长度的输出 $h(m)$，$h(m)$一般在 128 比特到 512 比特之间。Hash 函数要满足几个要求，其中最基本的要求是它必须是单向函数，即对给定的消息 m，$h(m)$很容易计算，而对给定的 x，则不可能求出满足 $x = h(m)$的 m。换句话说，单向函数就是容易计算但不可能求逆的函数，单向函数由此得名。

① 最早由 Leslie Lamport 提出的数字签名方案就是这种形式[21]。

在Hash函数的所有性质中，最常提到的是它的抗碰撞性，这是一个比单向性更强的要求。一个碰撞就是指两个不同的输入 m_1 和 m_2 满足 $h(m_1) = h(m_2)$。任意一个Hash函数都会有无穷多个这样的碰撞（因为有无穷多个输入而只有有限多个输出），因此Hash函数不可能是无碰撞的。抗碰撞性要求：尽管碰撞存在但不可能找到。

Hash 函数的抗碰撞性使得它可以用于数字签名方案，但有些具有抗碰撞的 Hash 函数却完全不适用于像密钥导出、单向函数等其他应用。实际上，密码系统的设计者希望 Hash 函数是一个随机函数，因此我们要求 Hash 函数与随机函数不可区分。其他定义都使设计者不再将 Hash 函数当做理想的黑箱子，而是要考虑 Hash 函数的性质与系统其他部分的相互作用。

定义 4：理想的 Hash 函数是从所有可能的输入值到有限可能的输出值集合的一个随机映射。

同理想的分组密码的定义一样（见 4.3 节），Hash 函数的这个定义也是不完整的。严格地说，没有像随机映射这样的 Hash 函数，我们只能考虑在所有可能的映射上的概率分布，但这对我们的目的来说已足够了。

现在我们再来定义对 Hash 函数的攻击。

定义 5：对 Hash 函数的攻击是一个非平凡的区分 Hash 函数与理想 Hash 函数的方法。

这里，理想的Hash函数必须与我们进行攻击的Hash函数有相同的输出。与分组密码一样，非平凡性要求只考虑所有常规的攻击，而且我们关于对分组密码的平凡攻击的说明也适用于 Hash 函数。

现在的问题是允许区分器进行多少计算。与分组密码不同，Hash 函数没有密钥，因此也没有像密钥穷举搜索这样的常规攻击。Hash 函数的一个重要参数是它的输出长度，它的一个常规攻击是寻找碰撞的生日攻击。对具有 n 比特输出的 Hash 函数，生日攻击大约需要 $2^{n/2}$ 步。生日攻击只对应于 Hash 函数特定的应用，在其他情况下，我们的目的可能是寻找一个 Hash 值的原像（给定 x，求满足 $h(m) = x$ 的 m），或者找到 Hash 输出中的某种结构，而一般的原像攻击大约需要 2^n 步。我们在这里不打算详细地讨论与哪种攻击相关和允许特定攻击区分器进行多少计算才算合理这些问题。显然，区分器必须比能得到相似结果的常规攻击更有效。这不是区分器的精确定义，我们也不知道应如何精确地定义它。如果有人声称找到了一个攻击，只要看他是否可以用不依赖于给定Hash函数特性的常规攻击得到类似或更好的结果即可。如果回答是肯定的，则这个区分器没有什么用，否则，这个区分器确实是一个有用的区分器。

与分组密码一样，我们允许指定一个相对低的安全指标。可以设想，一个512比特的Hash函数拥有 128 比特的安全性，在这种情况下，对区分器的限制应是 2^{128} 步。

6.2 实际的 Hash 函数

一个现实问题是好的 Hash 函数很少，于是人们不得不在 SHA 系列或 MD5 中选择。其他一些已公布的方案由于缺乏足够的关注和研究，目前仍然无法得到人们的信任。实际上，对 SHA 系列的分析也是不充分的，但它被 NIST 进行了标准化，NSA 对它做了进一步的完善①。

现实的Hash函数几乎都是迭代Hash函数，这里我们也只讨论这一类Hash函数。迭代Hash函数首先将输入分为固定大小的块序列，$m_1, \cdots, m_k$，这里最后一个块要进行填充以使其达到固定的长度，块的大小一般为 512 比特；然后使用一个压缩函数和固定大小的中间状态对这些消息块按顺序进行处理。这个过程从固定的 H_0 开始，可表示为 $H_i = h'(H_{i-1}, m_i)$，最后一个值 H_k 便为 Hash 函数的输出。

Hash函数的这种迭代结构在现实中很有意义。首先，与直接处理可变长度的输入相比，这样的迭代运算易于说明和实现；其次，这种结构使得我们可以在得到消息的第一部分时就开始计算，这样在实际应用中对一个数据流计算 Hash 值时，计算可以在传输中完成，从而不需要存储这些数据。

与分组密码一样，这里不打算用过多的篇幅详细地说明各种各样的 Hash 函数。Hash 函数的实现者们可以在我们引用的参考文献或因特网上找到更详细的说明，对其他人来说，不需要阅读完整细致的说明，这样做反而会使他们远离本书的主题。

6.2.1 MD5

由 Ron Rivest 提出的 MD5 是一个 128 比特的 Hash 函数[82]，它在 MD4 的基础上强化了抗攻击能力[79]②，是 MD4 的改进版。

计算 MD5 的第一步是将消息分为 512 比特的块序列，对最后一个块要进行填充，消息的长度信息也在其中。MD5 有 128 比特的状态变量，它们被分为 32 比特的 4 个字。压缩函数 h' 共有 4 轮，在每一轮中消息块和状态变量进行混合，这种混合由在 32 比特的字上的加法、异或、与、或、循环操作组合而成（详见[82]）。每一轮的整个消息块都混合在状态变量中，因此每个消息字都使用了4次。在4轮的压缩函数 h' 完成后，将结果与输入状态相加得到 h' 的输出。

在 32 比特字长上进行操作首先由 MD4 采用，由于这种操作方式在 32 比特的 CPU 上非常有效，现在已成为很多密码学基元的一个基本特征。

迭代 Hash 函数设计的一个基本思想是，如果是抗碰撞的，则由 h' 构造的 Hash 函数 h 也是抗碰撞的，h 出现碰撞可以归因于 h' 的碰撞。MD5 的问题是压缩函数 h' 有碰撞[20]。虽然 MD5 现在还没有已知的攻击，但压缩函数碰撞的存在使得我们对使用 MD5 有些担心。

---

① 不论对 NSA 的评价如何，他们所发表的密码学还是相当好的。

② MD4 非常快，但也很脆弱[25]，因此不宜使用。

对大多数应用来说，MD5 的 128 比特的长度是不够的。由生日悖论可知，对 Hash 函数大约进行 2^{64} 次计算就可找到一个碰撞，对现代的密码系统来说这是不够的，因此我们建议不要使用 MD5。

6.2.2 SHA-1

NSA 设计了安全 Hash 算法（The Secure Hash Algorithm），该算法已被进行了 NIST 标准化[71]。最早的版本叫 SHA（现在通常称为 SHA-0），它有一个缺陷，NSA 发现并修正了这个缺陷，NIST 公布了这个改进的版本，称为 SHA-1。他们一直没有发布这个缺陷的细节，三年后 Chabaud 和 Joux 找到了 SHA-0 的一个缺陷[16]，这个缺陷正是被 SHA-1 所修正的，因此有理由认为我们现在已经知道了事情的原委。

SHA-1 是 160 比特的 Hash 函数，与 MD5 一样也是基于 MD4 的，因此与 MD5 有很多共同的特征，但它的设计更保守。SHA-1 比 MD5 慢 2~3 倍，到目前为止，还没有发现 SHA-1 在安全性方面有什么问题，因此被人们广泛采用。

SHA-1 有由 5 个 32 比特的字组成的 160 比特的状态。与 MD5 一样，SHA-1 总共也有 4 轮，每一轮都由定义在 32 比特字长上的混合操作组成。所不同的是，MD5 对每个消息块进行 4 次处理，而 SHA-1 用线性递归方法将 16 个字长的消息块变为它所需要的 80 个字长。这是 MD4 技术的一般法则。在 MD5 中，消息的每一比特在混合函数中都被使用 4 次，而 SHA-1 的线性递归保证了消息的每个比特都至少影响混合函数 12 次。非常有趣的是，从 SHA-0 到 SHA-1 的惟一改变是对线性递归增加了 1 比特的轮转。

SHA-1 的主要问题是，它的输出长度为 160 比特，仅用 2^{80} 步就可以得到碰撞。这远低于密钥长度为 128~256 比特的现代分组密码的安全水平，也不能满足 128 比特的安全性设计要求。

6.2.3 SHA-256，SHA-384，SHA-512

最近，NIST 公布的标准草案里有 3 个新的 Hash 函数[75]，它们的结构与 SHA-1 类似。这 3 个 Hash 函数分别有 256 比特、384 比特、512 比特的输出，分别与密钥长度为 128 比特、196 比特、256 比特的 AES 一起使用。

这些 Hash 函数提出的时间不长，因此我们不想推荐采用它们，但除此之外，我们没有更多的选择。如果你想得到比 SHA-1 更强的安全性，那么就应使用有较长输出的 Hash 函数，但已公布的具有较长输出的 Hash 函数并没有受到广泛的关注，而 SHA 系列至少已被 NSA 仔细研究过，也许他们知道事情的真相。

SHA-256 比 SHA-1 慢许多。对较长的消息来说，用 SHA-256 计算 Hash 值大约与用 AES 或 Twofish 进行加密所需的时间相当，或许还略长一点。这个问题并不严重，因为我们以为计算 Hash 值比加密更难，因此我们不应对 Hash 函数比加密慢感到惊讶，相反我们对 SHA-1 和 MD5 的快速倒有点惊奇。不过对这些快速 Hash 函数攻击的研究很少，远不及对分组密码攻击的研究。

相对来说，SHA-384没有什么用处，计算SHA-384与计算SHA-512的运算一样，只是在最后舍弃了一些比特。实际上没有必要从中分出SHA-384这个函数，我们建议采用SHA-256或SHA-512。

6.3 Hash函数的缺陷

遗憾的是，所有的Hash函数都有一些不能满足我们的安全性定义的性质。

6.3.1 长度扩充

我们对这些Hash函数最大的忧虑是，它们都有一个易于避免但可以导致实际问题的长度扩充的缺陷。一个消息m被分成消息块序列$m_1, \cdots, m_k$，得到的Hash值为H。现在选择消息m'，它对应的消息块序列为$m_1, \cdots, m_k, m_{k+1}$，由于$m'$的前$k$个块与$m$的$k$个块相同，Hash值$h(m)$是在计算$h(m')$时的一个中间值，即有$h(m') = h'(h(m), m_{k+1})$。当使用MD5或SHA系列的Hash函数时，必须谨慎地选择包括填充和长度域的消息m'。由于构造这些域的方法是公开的，因此这不会成为问题。

长度扩充问题的存在是因为在Hash函数计算的最后缺少一个特殊的处理过程，结果导致$h(m)$恰好是计算$h(m')$时完成前k个块计算的中间状态。

对那些我们期望作为随机映射的Hash函数来说，这确实是一个令人惊奇的性质。事实上，这个性质使得前面提到的Hash函数都不能满足我们给出的安全性的定义。一个区分器只需构造一个合适的消息对（m, m'），然后检验这样的关系是否成立。对于理想的Hash函数，这样的关系一定不成立，因此这是一个有效的区分器。而且这样的攻击需要很少的计算，因此速度非常快。

这个性质有什么危害吗？设想在一个系统中，Alice给Bob发送一个消息m，用$h(X \| m)$作为对信息的认证，这里X是只有Alice和Bob知道的一个秘密信息。如果h是一个理想的Hash函数，这将是一个相当好的认证系统。但是由于长度扩充缺陷的存在，Eve现在可以在消息后面附加信息m，然后计算出与新消息相匹配的认证信息，于是在这个认证系统里Eve就可以修改消息，这样的系统对我们来说显然是没有任何用处的。

6.3.2 部分消息碰撞

第二个问题是具有迭代结构的Hash函数固有的缺陷，我们用一个特殊的区分器来加以解释。

任何区分器都首先要确定其区分Hash函数和理想Hash函数的设置，有时这样的设置很简单，如给定hash函数，寻找碰撞。下面我们使用一个略微复杂的设置。假定我们有一个系统用$h(m \| X)$作为对消息m的认证，其中X为认证密钥。攻击者可以选择消息m，但系统只能认证一个消息[①]。

① 大多数系统只允许认证一定数量的消息，这里是一个极端的情况。在现实生活中，许多系统里的消息都带有一个消息数，对这种攻击来说这与允许单一消息认证的系统一样。

对输出大小为 n 的理想的 Hash 函数，我们希望它具有 n 比特的安全性。攻击者选择消息 m 并使系统对其用 $h(m\|X)$ 进行认证，除了对 X 进行穷举搜索外，攻击者没有更好的方法。对迭代 Hash 函数，攻击者有更好的方法。他可以通过生日攻击用大约 $2^{n/2}$ 步找到两个消息 m 和 m' 使 Hash 函数发生碰撞，并使系统对消息 m 进行认证，然后用消息 m' 代替。由于 Hash 函数的迭代计算结构，只要出现了碰撞而且余下的输入一样，那么 Hash 值就一定相同，而消息 m 和 m' 有相同的 Hash 值，因此对任意的 X，有 $h(m\|X) = h(m'\|X)$。

这是区分器的一个典型例子。区分器设置自己的规则（实施攻击的设定），然后攻击这个系统，目的仍然是区分给定的 Hash 函数和理想的 Hash 函数。如果攻击成功，则这个 Hash 函数是迭代 Hash 函数；如果攻击失败，则它就是理想的 Hash 函数。

6.4 修正缺陷

我们希望能够得到可当做随机映射的Hash函数，但所有熟知的Hash函数都不具有这样的性质。在使用Hash函数时，我们是否必须克服长度扩充的缺陷呢？是否要制止部分碰撞的发生呢？除了这些，我们还需要克服其他的缺陷吗？

允许Hash函数有缺陷是一个极具危害的想法。可以肯定，存在的缺陷总会在某些应用中暴露出来，甚至对已被文献证明了的缺陷，在实际系统中也将不被禁止。即使可以很好地控制设计过程，也会不可避免地陷入到一个非常复杂的问题中。设想 Hash 函数有 3 个缺陷，而分组密码也有两个，签名方案本身有 4 个，那么你就必须检查由这些缺陷相互作用所产生的上百种可能的缺陷组合，分析它们实际上的不可能性。因此，我们必须对Hash函数本身进行修正。

我们必须指出，这是一个不很正统的观点。大多数密码学家乐意接受迭代Hash函数，甚至一些新的设计也没有对我们提出的问题进行修正。使用Hash函数时人们有时会尽力避免这些问题，但更多的时候则根本不去考虑它们，这也许会很幸运地避免了问题的出现，但实际上问题也会频繁地发生。在一个特殊基础上可以克服这些问题，但这不是解决问题的正确途径，我们应该对 Hash 函数本身进行修正。

6.4.1 一个彻底的修正方法

我们对如何修正Hash函数的文献不够了解，下面是我们在完成本书时提出的方法。Hash函数的这些问题可以用两次使用Hash函数的方法来避免，我们相信这种方法可以解决前面提到的所有问题，但设计Hash函数是很棘手的任务，我们还不能十分肯定这个结论是否正确。我们的解决方案还没有得到同行们的评价，这也许要花很多年的时间，我们希望能够引发人们对Hash 函数进行研究的兴趣。

设 h 是前面提到的某个 Hash 函数，我们用 $m \mapsto h(h(m)\|m)$ 代替 $m \mapsto h(m)$ 作为 Hash 函数[①]。我们将 $h(m)$ 放在要进行 Hash 运算的消息的前面，这可以保证 Hash 函数的迭代运算与消息的每一比特都有关，从而避免长度扩充和部分碰撞的攻击。

定义 6： 设 h 是一个迭代 Hash 函数，Hash 函数 h_{DBL} 定义为 $h_{DBL}(m) := h(h(m)\|m)$。

如果 h 是前面提到的任意一个 Hash 函数，这样构造的 Hash 函数 h_{DBL} 具有 n 比特的安全性，其中 n 是 Hash 函数的输出长度。

这个方案的缺点是比较慢，原因是必须对消息进行两次 Hash 运算，但我们对此并不介意，因为我们确信 Hash 函数的安全性远远比它的速度更重要。现在已有很多足够快的不安全系统，完全没有必要再设计一个这样的系统。然而，Hash 函数的速度是其应用性能的一个瓶颈，我们也希望能找到一个更好的 Hash 函数。

这个方案的另外一个缺点是需要存储整个消息，而不能在数据流的传输过程中计算其 Hash 值，一些需要这种能力的应用将不能使用 h_{DBL}。

6.4.2 一个更有效的修正方法

如何保持原来的 Hash 函数的速度呢？我们定义 Hash 函数为 $h(h(m))$ 而不是 $h(m)$，要求它的安全性为 $n/2$ 比特。通常，当碰撞攻击不可能时，我们希望 n 比特的 Hash 函数能为我们提供 n 比特的安全性[②]，而部分消息碰撞仅仅依赖于生日攻击，因此，如果将安全性要求降低为 $n/2$ 比特，那么这些攻击将被排除。

在大多数情况下，这种降低安全性的要求是不能被接受的。Hash 函数是针对碰撞攻击可能发生而设计使用的，因此就它们的大小来说已足够大了。如果将这种构造用于 SHA-256，得到的 Hash 函数将拥有 128 比特的安全性，这正是我们所需要的安全强度。

也许有人会对 n 比特的 Hash 函数仅提供 $n/2$ 比特的安全强度有争议，这种观点也是有根据的。如果没有明确说明这些事实，人们往往会滥用 Hash 函数并认为 Hash 函数能提供 n 比特的安全性。例如，人们对 AES 的 256 比特的密钥使用 SHA-256，认为它可以提供 256 比特的安全性。按照我们前面对分组密码安全性的解释，使用 256 比特的密钥只能得到 128 比特的安全性，这恰好与降低了安全性要求的 SHA-256 的修正版本相匹配。这并不是偶然的，因为在这两种情况下，值的大小与安全性强度之间的差异都归因于碰撞攻击，而我们总是假定碰撞攻击是可能的，这样就可以将值的大小与安全性强度很好地结合在一起。

下面是修正方案的更正式的定义。

定义 7： 设 h 是迭代 Hash 函数，Hash 函数 h_d 定义为 $h_d := h(h(m))$，其安全性强度为 $\min(k, n/2)$，其中 k 和 n 是 Hash 函数 h 的安全性强度和输出的大小。

① 记号 $x \longrightarrow f(x)$ 是不用函数名来表示函数的方法，如 $x \longrightarrow x^2$ 是将输入进行平方的函数。

② 甚至在 SHA-256 文档中也声称对 n 比特的 Hash 函数，寻找给定值的原像需要 2^n 步运算。

我们主要对 SHA 系列的 Hash 函数应用这种构造，对任意的 Hash 函数 SHA-X，其中 X 为 1、256、384、512 中的某一个，SHA_d-X 定义为将 m 映射为 SHA-X(SHA-X(m)) 的函数，例如，SHA_d-256 就是函数 $m \to$ SHA-256(SHA-256(m))。

这是一种新的修正方法，因此还不能取得人们的信任。幸运的是，我们可以证明修正的 Hash 函数 h_d 至少与基本 Hash 函数 h 一样安全①。HMAC 也使用类似方法抵抗长度扩充攻击。h_{DBL} 与 h_d 都消除了对实际系统构成极大威胁的长度扩充缺陷，而 h_{DBL} 是否拥有 n 比特的安全性有待于进一步证明，但可以肯定它们两个都有 $n/2$ 比特的安全性，因此在实际应用中可以使用更有效的 h_d。

6.5 Hash 的选择

现在还没有多少 Hash 函数可供选择，我们认为可以从 SHA_d 系列中进行选择，惟一需要考虑的是 Hash 函数的大小以及相关的性能。

前面已经指出，密码学界对 Hash 函数的研究很少。对分组密码经常采用区分器攻击，而对 Hash 函数的攻击类型主要是碰撞攻击，因此，没有人为了实用的目的采用我们的安全性定义对 Hash 函数进行分析。h_d 的构造使得我们可以采用 $n/2$ 比特的安全性，这不成问题。SHA 系列设计为抵抗碰撞攻击，因此至少也能达到这样的安全水平。

由于 80 比特的安全性对长期应用来说是不够的，因此不能使用 SHA_d-1，于是就只剩 SHA_d-256 和 SHA_d-512 可供选择（由于 SHA_d-384 与 SHA_d-512 有相同的计算量，SHA_d-384 不宜采用）。

我们已经确定了 128 比特的安全性设计标准，因此 SHA_d-256 是当然的选择。为了达到更高的安全性，可以选择使用 SHA_d-512。但是，如果认为这一定可以获得较强的安全性，那也许就错了。如果没有密钥长度超过 256 比特的好的分组密码，这样的安全性水平很难达到。当然，为 SHA_d-512 的那些比 SHA_d-256 好的性能所付出的代价也许很大，以至不能被人们接受，在这种情况下就只能接受具有 128 比特安全性的 SHA_d-256。

6.6 进一步的工作

关于 Hash 函数还需要做大量的工作。我们以为，如同我们在 20 世纪 80 年代对分组密码进行设计和攻击一样，密码学团体非常擅长于设计和攻击 Hash 函数。我们确实需要对 Hash 函数的设计和攻击进行更多的研究，一个好的开端是先研究 SHA 的缩减轮数的版本，在这个领域至今还没有取得进展。

① 我们在这里有点像在骗人。通过两次 Hash 运算，函数的值域缩小了，从而使生日攻击变得更容易了。

第7章　消息认证码

消息认证码（MAC，message authentication code）用于防止对消息进行篡改。加密使Eve不能获取消息的内容，但不能阻止Eve对消息本身进行各种操作，这时就需要消息认证码。与进行加密一样，MAC也需要一个Alice和Bob知道但Eve不知道的密钥K。当Alice向Bob发送消息m时，她不仅仅发送消息m本身，而且还发送一个由MAC函数计算得到的MAC值。Bob在收到消息及MAC值后，计算MAC值并与收到的MAC值进行比较，如果二者不相等，则丢弃这个不能通过认证的消息。由于Eve不知道密钥K，所以不能为篡改后的消息找到正确的MAC值，从而保证Eve不能对消息进行操作。

本章只讨论认证，有关加密与认证相结合的内容请见第8章。

7.1　MAC的作用

MAC是具有两个输入的函数，其中一个是固定大小的密钥K，另一个是任意大小的消息m。它产生固定大小的输出，即MAC值，我们将MAC函数表示为MAC(K, m)。为了对一个消息进行认证，Alice把消息m与MAC码MAC(K, m)一起发送。

我们首先只考虑MAC函数本身，但要注意，使用MAC函数很复杂，因为不仅仅是对消息应用MAC函数，我们将在后面的7.8节讲解这些问题。

7.2　理想的MAC

首先定义理想的MAC函数，然后将MAC函数定义为不能与理想的MAC区分的函数，而理想的MAC自然是一个随机映射。设n是MAC函数输出的大小，即输出结果的比特数。

定义8：理想的MAC函数是一个从所有可能的输入到n比特的输出的随机映射。

理想的MAC函数与理想的Hash函数类似，也是一个随机函数，主要的差别在于安全性的定义，对MAC函数的要求更宽泛一些，从而使MAC的计算比Hash函数更有效。

7.3　MAC的安全性

MAC有两个输入，一个是密钥K，另一个为消息m。攻击者不知道密钥K，更准确地说应该是攻击者不完全知道密钥K，因为系统的其他部分所具有的缺陷会给攻击者泄露密钥K的部

分信息。设k是攻击者掌握的关于密钥K的不确定性（uncertainty），这里也就是密钥K中攻击者不能确定的比特数。那么对攻击者来说，密钥K大约共有2^k个不同的值[①]。

定义9：设n是MAC函数的大小，k是密钥K的以比特为单位的不确定性（攻击者知道的）。对MAC函数的攻击是一个在$2^{\min(n,k)}$步内区分MAC函数和理想MAC函数的非平凡方法。

换句话说，MAC函数就是安全性限制为k比特的随机映射。这与一般的随机映射不同。对随机映射来说，攻击者什么也不知道；而对这样的MAC，攻击者不能确定K（如果可以确定，则$k=0$，而且安全性水平也为0，这时就没有什么需要讨论的了）。

当然，一个特殊的MAC函数可以指定比其输出大小低的安全性水平，这时攻击者的攻击应限制在$2^{\min(s,k)}$，其中s就是指定的安全性水平。关于K的不确定性，可以使MAC函数比Hash函数有更高的效率。

现在还没有人以这种方式来定义MAC函数的安全性，大多数研究者以很局限的方式定义MAC函数的安全性。粗略地说，定义使用的攻击模型是攻击者选择n个不同的消息，然后得到每个消息对应的MAC值，攻击者必须提出第$n+1$个与前面选择的n个消息不同的消息和与这个消息对应的有效的MAC值。这种攻击模型将一些攻击方法排除在外，如相关密钥攻击和攻击者知道密钥部分信息的攻击。这就是我们更喜欢我们这种安全性定义方式的原因。相比之下，我们的定义是健壮的，即使MAC函数被不恰当地使用或在一些特殊的环境里使用。

7.4 CBC-MAC

CBC-MAC是一种将分组密码转变为MAC函数的方法，所用的密钥K也就是分组密码的密钥。CBC-MAC所采用的方法是对消息m用CBC模式进行加密，而只保留最后一个消息块的密文，其余的全部丢弃。对一个分块为$P_1,\cdots,P_k$的消息，MAC函数的计算过程可以描述为：

$$\begin{aligned} H_0 &:= \mathrm{IV} \\ H_i &:= E_K(P_i \oplus H_{i-1}) \\ \mathrm{MAC} &:= H_k \end{aligned}$$

有时也会取最后一块的一半作为CBC-MAC函数的输出，这些可参阅相关的详细资料。

在一些教科书中，CBC-MAC的经典定义要求IV固定为0，而在CBC-MAC中使用CBC加密中提到的IV构造方法当然也是可行的，不过这样做似乎也没有什么好处。

① 不确定性（uncertainty）可以用熵来更精确地描述，但将会脱离我们现在的讨论。对那些喜欢了解细节的读者，k是攻击者知道的K的熵，以比特为单位。

除非对自己所做的一切很清楚，否则永远不要在加密和认证中使用同一个密钥。在CBC-MAC中，MAC值等于最后一个密文块，所以在CBC加密和CBC-MAC认证中使用相同的密钥特别危险。

使用CBC-MAC有一定的技巧，但一般认为如果所用的加密方案是安全的，则CBC-MAC就是安全的。CBC-MAC有许多碰撞攻击，这使得它的安全性只能是分块大小的一半[12]。一个简单的碰撞攻击为：设M是一个CBC-MAC函数，如果已知$M(a)=M(b)$，则知对任意的c都有$M(a\|c)=M(b\|c)$。CBC-MAC的这个性质源于它的结构，下面用c为一个消息块时的情形来说明这个性质，此时有

$$M(a \parallel c) = E_K(c \oplus M(a))$$
$$M(b \parallel c) = E_K(c \oplus M(b))$$

因为当$M(a)=M(b)$时，必有$M(a\|c)=M(b\|c)$。

攻击分为两个阶段。在第一阶段，攻击者收集大量消息的MAC值，直到产生碰撞，即在这个阶段攻击者得到了两个消息a与b且满足$M(a)=M(b)$。在第二个阶段，若攻击者得到了发送者对消息$a\|c$的认证，他就可以用$b\|c$代替消息$a\|c$而不用改变MAC值，这可以通过接收者对MAC值的验证，从而使接收者接到一个伪造的消息。（切记，在我们所处的模型中，攻击者生成一个消息并得到发送者对它的认证是可以接受的，因为在很多情况下这种事情都会发生。）这种攻击有很多的扩展变形，它们甚至对增加长度域和填充都有效[12]。

这是一个非平凡的攻击，因为它对理想的MAC函数不起作用。对理想的MAC函数，找到碰撞不成问题，这可以用同样的方法完成，但是当得到了满足条件$M(a)=M(b)$的两个消息a与b时，我们不能为一个新消息伪造一个MAC值，而在CBC-MAC中这是可以的。

已有一些理论研究的结果表明，当分组密码为128比特时，CBC-MAC有64比特的安全性[4]。这样的安全强度不能满足要求。如果我们使用256比特大小的分组密码，CBC-MAC的安全性将很好。

如何使用CBC-MAC也是一个需要注意的问题，不能直接对消息使用MAC函数，否则会导致简单的攻击存在。为此要按如下方法进行：

1. 从$l\|m$构造比特串s，其中l是消息m在固定长度格式下编码的长度。
2. 对s进行填充使得其长度为分组大小的整数倍（详见5.1节）。
3. 对填充后的s应用CBC-MAC。
4. 输出最后一个密文块或它的一部分，切记不要输出中间的值。

CBC-MAC的优点是，它与分组密码有相同的计算过程。加密和MAC是两个要应用于大量数据的函数，速度对它们来说都很重要，让它们拥有相同的基本函数有助于更有效地实现，特别是当用硬件实现时更是如此。

由于CBC-MAC很难正确使用，因此我们仍不愿使用CBC-MAC。

7.5 HMAC

理想的MAC是一个随机映射，而我们已经有了与随机映射类似的Hash函数，因此，很自然地会想到用Hash函数构造MAC函数，这正是HMAC所要完成的工作[3, 59]。HMAC的设计者当然很清楚第6章中介绍的关于Hash函数的问题。n比特的MAC函数希望能有n比特的安全性，而Hash函数对某些攻击来说只能保证$n/2$比特的安全性。如果使用标准的迭代Hash函数，将MAC(K, m)定义为$h(K\|m)$、$h(m\|K)$或$h(K\|m\|K)$中的任意一个都将是不安全的[77]，对$h(K\|m)$来说有长度扩充攻击，而对$h(m\|K)$的密钥恢复攻击大约需要$2^{n/2}$步。

HMAC的设计者对HMAC进行了精心的设计以抵御这些攻击，他们避免了攻击者可以得到密钥K的密钥恢复攻击，也避免了那些攻击者无需与系统进行交互的攻击。然而，由于可以利用迭代Hash函数的碰撞，生日攻击仍有效，因此HMAC仍然只有$n/2$比特的安全性。HMAC的结构使得攻击需要与系统进行$2^{n/2}$次的交互，这比在自己的计算机上完成$2^{n/2}$次计算要困难得多。

为了简单起见，我们不区分在线攻击和离线攻击，只关注攻击者要完成的工作量。对输出为n比特的Hash函数，即使进行了精心的设计，HMAC也仅有$n/2$比特的安全性。

当作为基元的Hash函数有一些我们不希望的性质时，HMAC会出现一些问题，有关HMAC的论文在这方面提供了很好的例子[3]。这就是为什么要强制为密码基元的特性提供一个清晰的说明的原因。

HMAC计算$h(K \oplus a\|h(K \oplus b\|m))$，其中$a$与$b$是指定的常量，即首先对消息进行一次Hash运算，然后再对结果和密钥进行Hash运算，详见文献[3, 59]中的说明。HMAC可以采用第6章中提到的任意一个迭代Hash函数。由于我们可以直接使用SHA系列而不必用SHA_d，因此对SHA_d是惟一的例外，但HMAC也考虑到了SHA_d解决的问题。

HMAC的结构清晰有效而且便于实现，因此我们很喜欢HMAC。以MD5和SHA-1作为Hash函数的HMAC应用很广，有关资料也很容易找到。为了获得128比特的安全性，我们愿意使用256比特的Hash函数，比如SHA-256。由于HMAC考虑到了SHA-256自身的缺陷，因此没有必要使用SHA_d-256来构造HMAC。

7.5.1 HMAC与SHA_d

只要是一个足够好的Hash函数，用$h(K\|m)$就可以得到一个好的MAC函数。我们认为SHA_d-256是一个好的Hash函数，于是我们可以得到一个好的MAC函数：

$$(K, m) \mapsto \text{SHA-256}(\text{SHA-256}(K\|m))$$

但是，我们推荐采用以SHA-256作为Hash函数实现的HMAC，它可以写为：

$$(K, m) \mapsto \text{SHA-256}((K \oplus a)\|\text{SHA-256}((K \oplus b)\|m))$$

第一个MAC很简单，为什么我们要推荐HMAC呢?

这是一个很好的问题，答案也很微妙。HMAC的设计考虑了不同攻击模型的攻击代价。假设Alice用MAC函数来认证，我们仅仅考虑攻击者必须完成的计算步数，而忽略了每一步计算是否需要与Alice进行交互。HMAC考虑到离线攻击（不需要任何交互）比在线攻击（需要进行交互）更易于完成，因此特别加强了对离线攻击的抵御能力，这就是为什么HMAC在第二次Hash运算时也使用密钥的原因。

当然，HMAC的设计者们的考虑是有道理的，离线攻击确实比在线攻击易于实现。但是，我们认为这并不说明离线攻击和在线攻击有不同的安全水平，问题是我们不打算给出代价因子，比如"在线攻击的一步相当于离线攻击的*S*步"，我们不知道*S*应为多少，因为我们并没有应用MAC的有关信息，因此，我们还坚持一贯的128比特的安全性水平。这不是对HMAC设计者们的批评，当他们设计HMAC的时候，大多数Hash函数都有128比特的输出。一个离线的碰撞攻击仅需要2^{64}步，这是必须要抵御的攻击；而对更大一点的Hash函数来说，这些攻击没有意义。

那么，究竟应该选择哪一个呢？我们认为应选择HMAC。HMAC额外的开销与SHA_d-256$(K\|m)$相比很小，并且HMAC得到了很充分的分析，积累了很多的经验。我们倾向于保守一些，而HMAC是一个更保守的选择。

7.6 UMAC

在MAC函数中，UMAC是一个如何利用密钥*K*的不确定性使得MAC比Hash函数更快的例子[8，60]①。UMAC可以比HMAC快一个数量级，对它的安全性也有一些证明。

遗憾的是，UMAC所具有的一些问题使得它与理想的MAC不同。

7.6.1 MAC的大小

UMAC的有关文献建议对大多数的应用使用64比特大小的结果。这是一个合理的实现标准，一两年以前我们也这样认为。MAC可以有比较小的结果是因为不能对MAC的结果进行离线的穷举搜索，如果对MAC值进行搜索，就必须与系统交互以验证这个MAC值是否正确。只要系统不允许攻击者伪造大量的消息，较小的MAC的结果也是安全的。

现在，我们改变了自己的观点，认为64比特的结果确实太小了，因为MAC的安全性不应依赖于系统的其他部分。作为MAC的设计者，我们不知道MAC将会被怎样使用，或者说滥用②。有些系统允许攻击者尝试许多伪造的消息，而其他的系统可能以不同的方式滥用MAC函数。我们不需要很复杂的设计，因为密码系统的设计本身已经很难了，让系统各部分

① UMAC的文件用"hash"这个词表示一致的Hash函数（universal hash function），这与我们所说的Hash函数完全不同，不要将二者混淆。

② 滥用是普遍存在的，如在IPsec的IKE协议中[41]，MAC函数有时在消息是秘密的而密钥是公开的情况下使用。

之间相互依赖使系统复杂化无疑是一种灾难。因此，我们要求 MAC 有 128 比特的输出。实际上，按照我们自己的设计准则 3（见 4.5.8 节），需要 256 比特的输出才能保证在生日攻击下仍有 128 比特的安全性。如果能够实现，就使用 256 比特的 MAC 值。但是，由于大多数系统通常使用 64 比特或 96 比特的 MAC 值，让人们接受需要 256 比特的值很困难。另一方面，由于每个消息要与它的 MAC 值一起发送，这样也将直接增加通信量。如果你拥有一个很好的 MAC 函数，而且不允许攻击者对 MAC 值进行生日攻击，那么可以放心地使用 128 比特的 MAC 值。

7.6.2 选择 UMAC

使用哪一个 UMAC 是一个十分重要的问题。最初的论文[8]定义了 4 种不同的 UMAC 函数：UMAC-STD-30、UMAC-STD-60、UMAC-MMX-30 和 UMAC-MMX-60。最新的 RFC 草案[60]提出了一种完全不同的构造方法，定义了另外两个函数：UMAC16 与 UMAC32。事实上，RFC 草案中的 UMAC 有 6 个参数，UMAC16 与 UMAC32 仅仅采用推荐的参数值集合。

我们可以只选择一个参数集合，是哪一个呢？参数对于 UMAC 的效率有很大的影响，有些在小端格式的机器（little-endian machine）上很快，有些在大端格式的机器（big-endian machine）上很快；有些非常适合采用 MMX 指令的 Pentium 机，而有些却更适合于有 32 比特乘法的其他的 CPU；有些使用无符号的乘法指令，而有些采用带符号的乘法指令。这种对照可以继续罗列下去，任意一个选择都一定会使你陷入对那些硬件缺陷的令人厌烦的讨论中。在任意的平台上，UMAC 的性能可以很容易地被 3 个到 5 个依赖于所选参数集合的因素改变，而且不同的平台需要不同的参数集合。

设计者应用极端的优化手段导致了过多的 UMAC 函数存在。为了适应对速度的要求，设计者针对不同的情况提出了不同的版本，因为对每一种情况都会有特殊的优化方法。

尽管这样做在短期内有一定的好处，但我们仍然不同意采用这种处理方法。UMAC 还没有一个标准，这与 HMAC 和 SHA-1 相似。不同的设计者选择 UMAC 的不同版本。这样如果需要与两个系统进行交互，就必须同时实现几个 UMAC 的版本。程序只允许在编译时对 UMAC 参数进行选择，如果在应用中需要两个或更多的 UMAC 版本，这将产生大量的问题。另外，系统 CPU 的结构在 20 年到 30 年间也会发生巨大的变化，从长远观点看，针对现在的结构进行的微优化（micro-optimizing）的想法是有缺陷的。

7.6.3 平台的灵活性

前面我们已经提到，为了适应特定的平台，各种各样的 UMAC 版本被优化。进一步说，UMAC 也对平台提出了迫切的要求，只要有快速的乘法指令和合理的内存量，UMAC 可以达到很高的速度。然而，我们认为 UMAC 对智能卡和其他拥有少量内存及 8 比特 CPU 的设备来说还是不能接受的。其临时的存储需求很高，而智能卡一般只有非常有限的内存量。

这些预示着UMAC只适合于台式机，而不适合于拥有小型CPU的设备[①]。我们猜测：与HMAC相比，若采用硬件实现，UMAC会相当昂贵。同样，我们认为UMAC不是一个好的标准，它更像是一匹赛马，而不是一匹劳作的马。它很快，但只在特殊环境下才有用。

7.6.4 分析的数量

UMAC最终的问题是缺乏对其设计的分析。前面已经提到，UMAC的安全性有一个证明，但这不能代替敌对的分析，一个可证明的安全系统被攻破也不是第一次了。当然，这是所有新设计共同的缺点，如AES、SHA-256以及我们在本书中提出的几个构造。在这些设计当中，AES可能受到了更多的关注。SHA-256由NSA设计，截止到目前来看它的设计很好。NSA雇用了很多专家，其中一些专家肯定在SHA-256发布前就对它进行了评审。我们在书中提出的一些新的构造还没有受到任何敌对的分析，但它们都是很保守的设计，至少与人们使用多年的标准构造一样好。UMAC是一个很积极的设计，它更多地依赖于对安全性的证明。我们乐观地假设UMAC的安全性证明有95%的可能是完备的和正确的，因此没有安全性证明的只有5%的可能。

7.6.5 为什么要讨论UMAC

如果我们不喜欢UMAC，为什么要用如此多的篇幅讨论它呢？我们认为UMAC的思想很重要，用密钥的不确定性提高MAC速度这个想法很有价值。我们愿意在这个领域里进行更深入的研究，一定有比Hash函数快得多而且与现代分组密码一样健壮和灵活的MAC。

7.7 选择哪一个MAC

从全面的讨论可以推断出我们主张选择HMAC-SHA-256——一个以SHA-256作为Hash函数的HMAC。很多系统使用64比特或96比特的MAC值，而我们确实想使用256比特的全部输出，为此有人会抱怨这样将增加总开销。对每个消息增加32字节（256比特）将不会在所有项目中受到欢迎。据我们所知，如果以传统的方式使用MAC，碰撞攻击就不存在，因此，我们希望将HMAC-SHA-256的结果截短为128比特也应该是安全的。

我们在此重申，在HMAC中没有理由使用SHA_d-256作为它的Hash函数，因为HMAC已经克服了长度扩充的缺陷。但是，为了获得128比特的安全性，我们需要使用256比特的Hash函数，这是HMAC-SHA-1不够理想的原因。

我们对目前的情况很不满足，相信构造更快的MAC函数应该是完全可能的。在合适的函数公布和分析之前，我们并没有多少工作可做。

---

① 有一个古老的争论，有些人认为小CPU设备是落后的，因为在未来的几年内它们会被大CPU所取代。我们不同意这种观点，小CPU经过数十年的时间积累了丰富的经验，这正可以移植到更小的应用中。20世纪70年代设计的8位的CPU仍然被广泛地使用着，尽管10多年前就有人预测它将会消亡。

7.8　MAC 的使用

恰当地使用 MAC 比直观上所感觉到的要复杂得多，下面我们将讨论这个问题。

当 Bob 收到 MAC(K, m)后，他就知道某个人知道密钥 K 并认可了消息 m。这样的声明隐含了所有的安全性要求，因此在使用MAC时必须格外小心。Eve可以记录从Alice到Bob的消息，然后在以后某个时间给Bob发送这个消息的一个副本。如果没有抵御这类攻击的措施，Bob将把这个消息的副本当做从Alice发来的有效消息而接收。如果Alice和Bob使用相同的密钥进行通信，类似的问题也会出现。Eve 可以将这个消息返回给 Alice，她就会相信这是 Bob 发来的消息。

在许多情况下，Alice 和 Bob 不仅想对消息 m 进行认证，而且要包括附加数据 d。这个附加数据包括防止重放攻击的消息数，还有消息源和目的地等，这些域（数据）通常是被认证的消息头的一部分。MAC 必须认证 d 和 m。一般的方法是将 MAC 函数用于 $d \| m$，而不是将 MAC 函数只用于 m。

下一个论点就是一个设计准则的内容。

设计准则 4： Horton 原则：认证消息的内容，而不是消息本身。

MAC 仅仅对一个字节串进行认证，而 Alice 和 Bob 想对具有特定含义的消息进行认证，二者的差别［即字节串本身（通信的内容）与字节串含义（通信内容的翻译）的差别］是很重要的。

假设 Alice 用 MAC 对 $m := a \| b \| c$ 进行认证，其中 a、b 及 c 是某些数据域，Bob 收到了 m 并将它也划分为 a、b 及 c。现在的问题是如何保证 Bob 也将 m 分成为同样的域，如果 Bob 采用的规则与 Alice 构造这个消息的规则不一致，Bob 就将得到错误的域值，于是 Bob 就会接收一个通过认证的假数据，这很糟糕。因此，Bob 必须得到与 Alice 相同的数据域。

这在一个简单的系统里不难做到，只要每个数据域都有固定的大小就可以了。但是，你很快就会遇到有些域的长度需要变化的情况，或者新版本的软件需要更大一些的域。当然，新版本的软件需要有向下的兼容性，这时仍有问题。一旦域的长度不再是常数了，Bob就应能够从内容来确定这些域，这时攻击者也就可以对内容进行篡改。例如，Alice 使用旧版本的软件和旧的比较短的域，而 Bob 使用新版本的软件，攻击者 Eve 篡改了 Alice 与 Bob 之间的通信，使得 Bob 相信新协议正在使用（实现的具体细节并不重要，MAC 系统不应依赖于系统其他部分的安全性），因此 Bob 用较大的域对消息进行分割，从而得到了假数据。

这种场合就需要 Horton 原理[93] ①，要认证消息所包含的内容而不是消息本身。这就意味着 MAC 认证的不仅是消息 m，而是 m 及 Bob 在解释时所需的所有信息，这应包括协议的标识

① 这个名字是按照儿童作家 Dr. Seuss 笔下的人物命名的[90]。

符、协议的版本号以及消息的标识符和各个数据域的大小等信息。一种方法是不用连接方式将各个域连接起来，而是用像XML那样的数据结构连接各个域，这样在解释时就不需要更多的信息了。

Horton原则说明了为什么低层的认证协议不能为高层的协议提供适当的认证。在IP包层的认证系统不能知道E-mail程序如何翻译数据，这就将它排除在检测消息的一致性之外。惟一的解决方案是，在低层进行认证外让E-mail程序提供自己的数据交换认证。

在进行认证时，总要仔细考虑哪些信息应该包含在认证的范围内，确保将包括消息本身在内的所有信息编码为一个字节串，而且编码应保证能以惟一的方式解析为数据域。这对我们在本章开始讨论的将附加数据与消息连接的方式也同样适用，如果对$d \| m$进行认证，最好能确定将连接后的消息$d \| m$分解为d和m的规则。

第8章 安全信道

我们最终又回到将要解决的第一个现实问题，安全信道可能是所有实际问题的共同点。

8.1 问题陈述

我们可以将这个问题定义为在Alice与Bob之间建立安全的连接，但这不是正式的定义。在弄清楚所讨论的问题之前，首先要对问题进行一定的形式化。

8.1.1 角色

首先，多数的连接都是双向的。Alice给Bob发送消息，Bob也给Alice发送消息。如果不想混淆这两个消息流，在协议中就要有某种不对称性。在实际系统中，也许一个是服务器，而另一个是客户，或者可以更方便地说一个是发起者（发起安全连接的一方），另一个是响应者。采用哪一种说法都可以，但必须将Alice和Bob的角色分配给问题中的双方，而且他们应该知道自己所扮演的角色。

当然，Eve总是存在的，他会用任意可能的手段对这个安全信道进行攻击。Eve可以阅读Alice和Bob之间的所有通信，并且可以任意修改这些通信的内容，特别是Eve可以对通信数据进行删除、插入、修改等操作。

我们可以把Alice向Bob发送消息想像为两台计算机通过某种网络互相发送消息。另一个有趣的应用是可以安全地存储数据，如果认为存储数据是向“未来”发送数据，那么所有的讨论都有意义。Alice和Bob可以是同一个人，信息传递的媒介可以是磁带，即使这样，仍然想保护媒介使其不被窃取和修改。当然，如果你向“未来”发送数据，就不能有交互的协议，因为“未来”不会向“过去”发送数据。

8.1.2 密钥

要实现安全的信道，我们需要一个共享的密钥。我们假定Alice和Bob有一个共享的秘密密钥K，其他任何人都不知道这个密钥。这是必要的条件。密码学基元永远不把Alice当做一个人来识别，他们至多只识别这个密钥，因此，Bob的验证算法只能告诉他这样的事实：这个消息是由知道密钥K、充当Alice这个角色的某个人发出来的。只有当Bob知道这个密钥K只限于他和Alice知道时，这样的事实对Bob来说才有意义。

如何建立密钥不是这里所关心的问题，我们假定密钥已经建立，关于密钥的管理将在第15章详细讨论。对密钥有如下的要求：

- 只有 Alice 和 Bob 知道密钥 K。
- 每次安全信道被初始化时，就会从密钥 K 产生一个新的值。

这里的第二条很重要。如果相同的密钥反复使用，前一次会话的消息就可以被Alice或Bob重放，这会引起混乱。因此，即使以固定的口令为密钥时，在 Alice 与 Bob 之间仍然需要一个密钥协商协议来建立惟一合适的密钥K，在每次建立安全信道时都必须重新运行这个协议。每次会话所用的密钥 K 称为会话密钥，如何产生将在第 15 章讨论。

安全信道的设计要达到 128 比特的安全性，按照我们的设计准则（见 4.5.8 节），则需要 256 比特长的密钥 K，因此为 256 比特的值。

8.1.3 消息或字节流

另一个问题是，我们是将 Alice 和 Bob 之间的通信当做离散消息的序列，还是作为连续的字节流。我们仅考虑处理离散消息的系统，它可以很容易地用来处理字节流，这只需将数据流分割为离散的消息，在接收端再组合成字节流就行了。实际上，几乎所有的系统在密码层都使用离散消息系统。

我们还假设实现在 Alice 和 Bob 之间通信的底层传送器不可靠。从密码学的观点来看，即使是一个可靠的通信协议，如TCP/IP，也不能构成一个可靠的通信信道，因为攻击者可以在不打乱数据流的情况下在TCP流中改变、移动或插入数据。对数据包丢失这样的随机事件来说，TCP是惟一可信任的，它不能抵御实施主动攻击的敌手。从敌手的观点来看，根本没有可靠的通信协议（这是一个密码学家为什么会用不同方式看这个世界的好例子）。

8.1.4 安全性

我们现在可以明确地表示信道的安全性。Alice 要发送消息序列 $m_1, m_2, \cdots$，安全信道算法对消息序列进行处理并发送给 Bob，Bob用安全信道算法处理收到的消息序列，最后得到消息序列 $m'_1, m'_2, \cdots$。

以下的性质必须满足：

- 除了消息 m_i 的大小和发送时间外，Eve 不能得到其他任何有关消息 m_i 的信息。
- 即使Eve进行主动攻击，即对正在进行通信的数据进行操作，Bob收到的序列 $m'_1, m'_2, \cdots$ 一定是序列 $m_1, m_2, \cdots$的子序列，而且 Bob 清楚地知道他收到的是哪一个子序列（子序列就是从原序列中去掉 $s(s \geq 0)$个元素后得到的序列）。

第一个性质是保密性。在理想状态下，Eve不应该得到关于这些消息的任何信息。在现实中要隐藏消息的大小和时间等信息十分困难，已知的解决方法要求Alice用她所使用的最大带宽发送连续的消息流，如果她没有消息要发送，就必须生成一些平凡的消息进行发送。对军事应用来说，这也许是可以接受的，但对大多数的民用项目来说，这是不能接受的。假设Eve在

通信信道上看到了消息的大小和时间，她就可以知道通信的对象、通信量和通信的时间，这些统称为流量分析（traffic analysis）。流量分析可以产生很多信息，而且很难阻止。我们在本书中不解决这个问题，因此Eve可以在安全信道上完成流量分析。

第二个性质保证Bob只得到适当的消息，并且是按照正确的顺序。在理想状态下，我们希望Bob收到的消息序列恰好是Alice发送的消息序列，但现实中没有在密码学意义上可信的通信协议，Eve总是可以在传输过程中删除一个消息。由于我们不能阻止消息的丢失，Bob就必须设法处理仅得到一个子序列的情况。注意，Bob收到的消息序列的顺序是正确的，并且没有重复的消息，也没有被修改的消息及Alice以外的人发送的假消息。更进一步说，要求Bob清楚地知道他丢失了哪些消息。这些对有些应用程序很重要，因为在这些程序中消息的翻译依赖于收到它们时的顺序。

在大多数情况下，Alice都想确保Bob可以得到她发送的所有信息。方法是Bob对收到的消息给Alice发送确认（直接的或隐含的），同时Alice重新发送那些没有得到Bob确认的消息。很多系统都能够实现这样的方案。安全信道不可能主动完成消息的重发，因此消息的重发必须由Alice自己完成，或者由使用安全信道的协议层来完成。

为什么不用安全信道内部的重发功能使得安全信道成为可信信道呢？原因是这样会将安全信道的描述复杂化，而我们希望以安全性为主导的模型应该简单。消息的确认和重发是标准的通信协议技术，它们可以在安全信道的上层实现。这是一本关于密码学的书，而不是关于基本通信协议技术的书。

8.2 认证与加密的次序

显然，我们将对消息应用加密和认证两个函数，有两种方法：一种是首先进行加密，然后再对密文进行认证；另一种是首先进行认证，然后再对消息和MAC值进行加密。

支持先进行加密的主要理由有两个。一个是理论结果表明，根据安全加密和认证的特定定义，先加密的方案是安全的，而先认证的方案是不安全的。进一步的研究表明，如果加密方案有一个特殊的缺陷，先认证的方案就不安全。在实际系统中，我们永远不会使用有这种缺陷的加密方案。然而，这些较弱的加密方案满足一个特殊形式的安全性定义（这是一个说明实际安全性和可证明安全性的差别的好例子）。对这个弱的加密方案的密文应用MAC就修正了这个缺陷，从而使其变得安全了。对任意的实际加密方案来说，这样的理论结果没有什么意义。事实上，有类似的证明表明，这些问题在流密码（如CTR模式）和CBC模式的加密方案中不可能发生。

先进行加密的第二个理由是，它在处理假消息时更有效。对正常的消息，不管是哪种次序，Bob都必须进行消息解密和检查认证。如果消息是伪造的（即有错误的MAC域），Bob最终将会丢弃。在先加密的方案中，Bob永远不会对一个假消息解密，因为在解密之前就可以识别这个假消息，从而将它丢弃。但在先认证的方案中，Bob首先必须对消息进行解密，然后再进行

认证检查，这样对假消息处理就需要做更多的工作。当Eve给Bob发送大量的假消息时，多做的工作不可小视。在先加密的方案中，Bob节省了解密的时间，从而减轻了CPU的负担。在一些特殊环境里，这可以使拒绝服务（DOS, denial-of-service）攻击变得更困难。现实中的很多情况是，DOS攻击是通过使通信信道饱和来进行攻击的，而不是让Bob的CPU陷入困境。我们不一定要为此找到依据，因为我们愿意牺牲效率以换取安全性。

支持先进行认证的主要理由也有两个。其一是在先加密的结构中，Eve可以看到MAC的输入和MAC值；而在先认证的结构中，MAC的输入和真正的MAC值被隐藏，Eve只能看到密文和加密后的MAC值。这个性质使得对MAC进行攻击在先认证的情况下要比在先加密的情况下难得多。实际中的攻击选择的是两个函数（加密和认证）中最后应用的那一个。如果最后进行的是加密，Eve就设法攻击加密函数；如果最后进行的是认证，Eve就设法攻击认证函数。在一般情况下，认证比加密更重要，因此，我们更愿意让加密函数处在Eve的直接攻击之下，从而尽可能地保护MAC。

认证要比加密更重要，这与很多人的认识相反，但这个观点的确是正确的。设想正在使用的是安全信道，分别考虑在知道通信内容和可以对通信数据进行修改这两种情况下，Eve的攻击所带来的最大损害。由此可以发现，在大多数情况下，对通信数据进行修改是毁坏性的攻击，远比看到通信内容更具有破坏性。

其二，先认证符合Horton原则。按照Horton原则，要对消息的内容进行认证，而不是对消息本身进行认证。对密文进行认证破坏了这个原则，从而就产生了一个缺陷。危险来自于密文可能通过了Bob的认证检测，因为他可能使用了与Alice加密所用的不同密钥。于是，尽管通过了认证检测，Bob仍将得到一个与Alice所发送的消息不同的明文。有一个具有特殊结构的IPsec就有这样的问题[33]。这个缺陷必须被修正，而且可以通过将加密密钥包括在需认证的附加数据中来完成。但是，我们不愿在加密以外的场合使用密钥，这会引发另外的危险，而且人们也不想让一个有缺陷的MAC函数泄露加密密钥的有关信息。标准的解决方案是从一个安全信道密钥导出加密密钥和认证密钥，正如我们在8.4.1节中所做的那样。这种方法可以克服这个缺陷，但会导致一个交叉依赖关系，认证会依赖于密钥导出系统。

论证哪种操作顺序更好需要花些时间。两种方案均可造就一个好的系统，也可以产生一个坏的系统。每种操作各有其优缺点，采用哪种操作取决于更侧重什么。我们选择认证优先，因为安全性和简单性对于节省CPU时间来说更重要。

8.3 概述

解决方案由三部分组成：消息编号、认证和加密。

8.3.1 消息编号

消息编号十分重要，它们可以用于为加密算法提供需要的IV；也可以让Bob拒绝重放的消息，同时不必保存一个大数据库；它们可以告诉Bob在通信过程中丢失了哪些消息；它们也

能保证Bob以正确的顺序接收消息。为此，消息号必须单调增加（较后的消息必须有较大的消息号）而且是惟一的（没有两个消息的消息号相同）。

为消息指定消息号很容易。Alice将第一个消息编号为1，第二个消息编号为2，依次类推。Bob记录他收到的最后一个消息的编号。任意一个新消息对应的消息号要比以前的消息号大，Bob靠接收这样递增的消息号就可以肯定Eve不能给他重发旧消息。

在我们的安全信道设计中使用32比特的消息号。第一个消息编号为1，消息的最大编号为 $2^{32}-1$。如果消息号溢出，Alice就必须停止使用现在的密钥，并且再一次运行密钥协商协议生成新的密钥。消息号必须是惟一的，所以不允许将它重新置0。

我们可以使用64比特的消息号，但这会增加开销（我们为每个消息使用8个字节的消息号，而不是4个字节）。32比特可以满足大多数的应用，而且密钥也应该经常更换①。当然，只要愿意，也可以使用40比特或48比特的消息号，这些都无关紧要。

大多数C程序员都喜欢从0开始计数，而消息编号为什么要从1开始呢？这是一个简单的实现技巧。如果有 N 个数可以被分配，Alice和Bob就要有能力记录 $N+1$ 个状态，毕竟截止到某一时刻，消息号可以是$\{0, \cdots, N\}$中的任意一个。限制消息数量为 $2^{32}-1$，则状态可以用32比特的数来表示。如果我们从0开始编号，那么在每次实现时，就要有一个额外的标志来说明是没有消息发送还是消息编号被用完。额外的标记增加了大量很少被执行的编码。如果它几乎不被使用，则将只会被测试几次，因此可能会不起作用。这是另一个容易出错的地方，我们可以通过从1开始计数来避免它。

在本章以后的内容中，我们将用 i 代表消息编号。

8.3.2 认证

认证函数需要一个MAC。可能与你期望的一样，我们将使用HMAC-SHA-256和它完整的256比特的输出。MAC的输入由消息 m_i 和附加的认证数据 x_i 组成。正如我们在第7章中解释的那样，一些有关的数据必须包含在认证中，Bob将用这些数据解析这个消息。这些数据应包括协议的版本号、域的大小等。我们用的是安全信道，因此 x_i 必须由其他程序来提供。在我们看来 x_i 就是一个比特串，而且Alice和Bob应有相同的 x_i。

设 $l(\cdot)$为一个返回数据串的字节长度函数，MAC中的值可以计算如下：

$$a_i := \mathrm{MAC}(i \| l(x_i) \| x_i \| m_i)$$

其中 i 与 $l(x_i)$都是32比特的无符号整数（即最低有效字节在前的格式）。$l(x_i)$保证了 $i \| l(x_i) \| x_i \| m_i$ 惟一地分解成它对应的域，要是没有 $l(x_i)$，结果是将会有许多方式将它分割为 i、x_i 和 m_i。当然，对 x_i 进行编码的方式应该使它不需要更多的信息就可以分解为不同的域，但是这不是我们在这个水平上可以保证的事，使用安全信道的程序必须保证这一点。

① 所有的密钥要在合理的时间间隔内更新，频繁使用的密钥更要经常更新，限制一个密钥最多处理 $2^{32}-1$ 个消息是合理的。

8.3.3 加密

为了加密，我们将使用CTR模式下的AES，消息号为CTR模式提供它所需要的惟一的瞬时值。

我们将每个消息的大小限制为不超过$16 \cdot 2^{32}$个字节，即块计数限制为32比特。当然，我们也可以使用64比特的计数，但是32比特的计数在许多平台上都更易于实现，而且对大多数应用来说也不需要处理这么大的消息。

密钥流由$k_0, k_1, \cdots$组成，对瞬时值为i的消息，密钥流定义为：

$$k_0, \cdots, k_{2^{36}-1} := E(K, 0 \| i \| 0) \| E(K, 1 \| i \| 0) \| \cdots \| E(K, 2^{32}-1 \| i \| 0)$$

这里每一个明文块由32比特的块编号、32比特的消息编号以及64比特的0组成。密钥流很长，我们仅仅使用最初的$l(m_i)+32$字节（不必计算密钥流的其他字节），连接m_i与a_i，并与$k_0, \cdots, k_{\ell(m_i)+15}$进行异或运算。

8.3.4 格式结构

我们不能直接发送加密后的$m_i \| a_i$，因为Bob也要知道消息号。实际发送的消息i由32比特的编码（最低有效字节在前）与加密的m_i和a_i组成。

8.4 详细说明

现在讨论安全信道的具体细节。为了方便起见，我们认为信道是双向信道，因此相同的密钥可以在两个方向上使用。如果我们将信道定义为单向信道，可以肯定有人在两个方向使用相同的密钥，最终破坏了安全性。采用双向信道可以降低这样的危险。

我们用伪代码来描述算法，对熟悉传统程序设计的人来说这样的描述很容易阅读。程序块用缩进和成对的关键词（如if/fi，do/od）表示。

8.4.1 初始化

我们给出的第一个算法是信道数据的初始化算法，它有两个主函数：建立密钥和建立消息号。我们从信道密钥导出4个辅助密钥：Alice给Bob发送消息所用的加密密钥和认证密钥，Bob给Alice发送消息所用的加密密钥和认证密钥。

```
function InitializeSecureChannel
input:  K    256比特的信道密钥
        R    角色说明，确定参与方是Alice还是Bob
output: S    安全信道的状态
```

首先计算所需要的4个密钥是不带长度或零终止的ASCII串。

KeySendEnc ← SHA_d-256(K‖ "Enc Alice to Bob")

KeyRecEnc ← SHA_d-256(K‖ "Enc Bob to Alice")

KeySendAuth ← SHA_d-256(K‖ "Auth Alice to Bob")

KeyRecAuth ← SHA_d-256(K‖ "Auth Bob to Alice")

如果是Bob，交换加密密钥和解密密钥。

if R = "Bob" **then**

  swap(KeySendEnc, KeyRecEnc)

  swap(KeySendAuth, KeyRecAuth)

fi

将发送和接收的计数器置0。发送计数器是发送的最后一个消息的编号，接收计数器是接收的最后一个消息的编号。

(MsgCntSend, MsgCntRec) ← (0, 0)

状态打包。

S ← (KeySendEnc,
    KeyRecEnc,
    KeySendAuth,
    KeyRecAuth,
    MsgCntSend,
    MsgCntRnc)

return ***S***

还有一个用于擦去状态信息S的函数。对这个函数我们将不做详细说明，它的功能就是清除存储状态信息S的内存。由于密钥在这里存储，清除这个区域的信息是必要的。在许多系统中，仅仅释放内存空间而不清除内存中的内容，因此，你在使用完后必须擦掉S。

8.4.2 发送消息

现在开始考虑发送消息的处理。这个算法以会话状态、要发送的消息以及用于认证的附加数据为输入，输出准备发送的加密和认证消息。接收者必须拥有相同的附加数据以进行认证检查。

function SendMessAge

input: S 安全会话状态

  m 要发送的消息

  x 用于认证的附加数据

output: t　　发送给接收者的数据

首先检查消息编号并更新。

assert MsgCntSend < $2^{32}-1$

MsgCntSend ← MsgCntSend + 1

i ← MsgCntSend

计算认证，$l(x)$和 i 都用 4 字节编码，最低有效字节在前。

a ← HMAC-SHA-256(KeySendAuth, $i \| l(x) \| x \| m$)

$t \leftarrow m \| a$

生成密钥流。明文块是由 4 字节的块编号、4 字节 i 以及 8 个全 0 字节组成，整数的最低有效字节在前。E 是 256 比特密钥的 AES 加密方案。

K ← KeySendEnc

$k \leftarrow E_K(0\|i\|0) \| E_K(1\|i\|0) \| \cdots$

形成最终的文本。i 用 4 字节编码，最低有效字节在前。

$t \leftarrow i \| (t \oplus \text{First-}l(t)\text{-bytes}(k))$

return t

算法可由我们前面的讨论直接得到。我们要检查消息计数，这个检查很重要。如果计数器发生轮回，整个安全性都将崩溃，这是一个我们经常看到的错误。认证和加密正如我们前面说明的一样。最后，我们将 i 与加密和认证的消息一起发送，这样接收者也就知道了消息编号。

我们注意到由于 MsgCntSend 被修改了，因此会话状态要更新。需要再次强调的是：这是必需的，因为消息号必须惟一。事实上，几乎所有的工作对安全性来说都是必需的。

8.4.3 接收消息

接收算法需要 SendMessage 的输出和相同的认证附加数据。

function ReceiveMessage

input: S　　安全会话状态

t　　接收者收到的数据

x　　用于认证的附加数据

output: m　　实际发送的消息

接收者收到的信息必须至少包含有 4 字节的消息号和 32 字节 MAC 域，这个检查保证以后对消息的分割可正常进行。

assert $l(t) \geq 36$

从 t 分割出 i，分割很容易，因为 i 占用 4 个字节。

$i \| t \leftarrow t$

与发送者一样生成密钥流。

$K \rightarrow$ KeyRecEnc

$k \rightarrow E_K(0\|i\|0) \| E_K(1\|i\|0)\|\cdots$

解密消息和MAC域并进行分割。分割是确定的，因为 a 总是占用32个字节。

$m\|a \leftarrow t \oplus$ First-$l(t)$-Bytes(k)

重新计算认证。$l(x)$和 i 都用4字节编码，最低有效字节在前。

$a' \leftarrow$ HMAC-SHA-256(KeyRecAuth, $i\|l(x)\|x\|m$)

```
if a' ≠ a then
    destroy k, m
    return AuthenticationFailure
else if i ≤ MsgCntRec then
    destroy k, m
    return MessageOrderError
fi
MsgCntRec ← i
return m
```

我们使用规范的操作命令。你可以在解密之前对消息编号进行检查，但是，如果 i 在传输过程中被损坏了，这个函数将会返回一个错误的出错信息。本应返回消息被损坏的出错信息，它却返回了消息次序错误这样的出错信息。由于呼叫者对这两种情况可能采用不同的方式处理，因此给呼叫者返回的信息不应有错。有人喜欢先进行检查，因为这样可以更快地处理假消息。但我们认为这并不很重要，如果你收到了太多的假消息以至于处理它们的速度变得非常可观，则说明已经遇到了更大的问题。

有一个问题对接收者很重要。在认证检测通过之前，函数ReceiveMessage不能泄露关于密钥和明文的任何信息。如果认证失败，返回一个出错指示，但密钥流和明文不能被泄露。在实现时要清除存放这些信息的内存。为什么说这很重要呢？因为总假定攻击者知道密文，因此明文消息会泄露密钥流。危险在于攻击者可以发送一个伪造的消息（带有不正确的MAC值），但他可以从接收者放弃的数据中得到密钥流。这是一个极端的模型，总假定程序丢弃或泄露的任何数据都会被攻击者获得。在返回错误信息之前销毁数据 k 和 m，这个程序就可以保证永远不泄露这些数据。

8.4.4 消息的顺序

与在发送端一样，接收者要修改MsgCntRec，即要更新 S。接收者要确认接收的消息编号严格地递增，这可以保证任何一个消息都不会被接收两次，但如果消息流在传送时改变了顺序，一些有效的消息将会丢失。

这个缺陷很容易克服，只是要付出一定的代价。如果要让接收者接收次序颠倒的消息，可以利用安全信道的应用程序处理次序颠倒的消息。很多应用程序都不能处理这个问题，而有些应用程序被设计成可以处理这样的情况，但当消息次序颠倒时，会有很微妙的隐错（与安全性有关）。在大多数情况下，我们宁愿修正传输层来阻止消息次序颠倒的情况发生，因此，安全信道不必处理这些事情。

我们知道的另外一种情况是，接收者允许消息以任意次序到达。对IP包进行加密和认证的IP 安全协议IPsec[51]就是这种情况。因为在传输中IP包可以重新排序，而且使用IP的程序也知道这个性质，IPsec不仅仅保存收到的最后一个消息的编号，而且要保留一个重放保护的窗口。如果c是收到的最后一个消息的编号，那么，IPsec就为消息号$c-31, c-30, c-29, \cdots, c-1, c$分别保留一个位映像，其中的每一位都显示与这个消息号所对应的消息是否被收到。编号比$c-31$小的消息总是被拒绝，而对编号在$c-31$与$c-1$之间的消息，只要对应的比特为0就被接收。如果新消息的编号比c还大，更新c，位映像移位保持不变。这样的位映像结构能使在不增加太多接收者的状态下，允许消息在有限范围内的次序发生颠倒。

8.5 选择

我们给出的安全信道的定义并不实用，特别是在嵌入式硬件中实现安全信道时，实现SHA-256的开销相当大。此外，我们还有哪些选择呢?

我们在编写这本书的时候，Niels 遇到了一个有这样问题的客户。他原计划使用OCB[83, 84]，这个模型的优点是加密和认证都使用一个分组密码方案，而且不需要其他的密码基元。遗憾的是，围绕OCB的专利问题使它成为一个高风险的昂贵的方案。最近提出的分组密码模式同样可以提供加密和认证功能，而且专利方面的问题也相对简单。

为了解决这个问题，Doug Whiting、Russ Housley 及 Niels 研究出一个将CTR 模型的加密和CBC-MAC的认证相结合的方案，这个加密模式又称CCM[94]。与OCB相比，CCM需要两次同样多的计算去加密和认证一个消息，但据我们所知CCM还没有专利发布，而且设计者也知道没有专利涉及CCM，他们没有也不会申请专利。Jakob Jonsson 给出了CCM安全性的一个证明[44]，而且NIST 正在考虑是否将CCM 作为分组密码模式的标准。

那么，究竟应该使用OCB和CCM中的哪一个呢？两个都是比较新的模式（CCM更新），这使人们对使用它们有些疑虑，我们并不完全信任这两个模式的安全性证明。CCM是一些被认可的技术组合，而OCB完全是一种新的构造。OCB的效率两倍于CCM，但它的使用被多方面的专利所限制。

两种模式的主要差别是在碰撞攻击下的不同表现。CCM使用CTR进行加密。在前面我们已经讲过，当加密的消息块超过2^{64}个时（假定块的大小为128 比特），CTR将会泄露信息。因此，CCM限定一个密钥可加密的消息块数为2^{60}，这样可以将信息的泄露减少至可忽略的水平

（小于1比特）。对CCM的认证部分还没有已知的碰撞攻击。尽管CCM不能获得128比特的安全性，但也已经很接近了，而且它至少与其他分组密码模式有同样好的性能。

另一方面，OCB有一个认证的碰撞攻击。如果发生了碰撞，认证的性质就完全没有了[35]，这意味着OCB不能获得128比特的安全性。如果每个密钥加密了2^{48}个消息块，它仅仅获得了大约81比特的安全性。尽管现实中的很多系统都使用类似的安全性，我们仍然认为这不够好。

我们更喜欢CCM，我们认为它是本章安全信道的一个合理的选择。我们也许有偏见，因为我们参与了CCM的设计。

一个重要的观点是：CCM、OCB以及类似模式都不能提供完全的安全信道，它们提供了加密和认证功能，每一个数据包都需要密钥和惟一的瞬时值。修改我们的算法使其利用这些分组密码模式而不是分离的MAC和加密函数是很容易的事。这时只需要两个密钥，而不是InitializeSecureChannel中的4个辅助密钥。每个方向有一个密钥，而瞬时值可以通过对消息号进行填充来构造。

8.6 结论

安全信道是密码学最有用的应用之一，几乎所有的密码系统都要使用。如果给定性能良好的加密和认证函数，建立安全信道不是很困难，只是有很多细节需要注意并要加以正确处理，这是在密码学中的常见情况。在现实中，主要困难不是实现安全信道系统，而是让人们相信他们需要安全信道和建立密钥。

第9章 实现问题（I）

本章我们将讨论有关实现的问题。实现密码系统与实现普通的程序不同，应该有一些不同的处理手段。

这里，最大的问题仍是最弱环节性（weakest-link property，见2.2节），它使得现实的安全性变得很复杂。事实上，实现错误（通常是缓存器溢出）截至目前为止仍然是一个实际系统中最大的安全性问题。如果你在最近几年里一直关注计算机安全问题，就会知道这个实际情况。实际上，很少听说一个密码系统被攻破，这不是因为密码系统已足够好了（我们的分析足以说明这一点），而是因为寻找与实现有关的漏洞比寻找密码系统的弱点更容易。聪明的攻击者不会为密码学费心，因为有比这更简单的途径。

到现在为止，本书仅局限在对密码学的讨论上，本章我们将关注密码系统的运行环境。系统的每一部分都会影响它的安全性，为了很好地完成一项工作，整个系统的设计要从最基础开始，并将安全性作为主要目标，而不仅仅是在设计时注意安全性。我们所说的系统是大系统，任何有可能损害系统安全性的因素都将包括在内。

操作系统总是最主要的部分，但目前广泛使用的操作系统都不是以安全性为主要目标的。由此得出的结论就是，要实现安全的系统是不可能的。我们不知道如何实现安全系统，也不知道有谁能实现这样的系统。现实中的系统包括很多并不是为安全性而设计的成分，因此，要达到我们实际所需要的安全强度是不可能的。那么，我们就只有放弃了吗？当然不是。当我们设计密码系统时要尽最大努力，至少保证我们的这一部分是安全的，这有一点儿各人自扫门前雪的意味。尽管我们不能做什么，仍然要考虑系统的其他部分。这就是我们写本书的目的：让人们认识到安全性的隐患及其重要性。

我们在前面已经提到了要有一个有效的密码系统的理由，因为对密码系统进行攻击有很大的危害性，这种攻击是不可见的。攻击者已经成功地攻破了你的密码系统，而你可能一点儿也不知道，这就好比一个小偷拥有你家房子的一串钥匙，如果这个小偷行动谨慎，你怎么能发现他呢？

我们的长期目标是建立安全的计算机系统。为了达到这个目的，每个人都必须做好自己的工作。我们的任务（也是本书的内容）就是使密码系统安全。系统的其他部分也必须安全，但我们现在无能为力，也许其他人可以完成这个任务，或许将来我们也能。到目前为止，系统整体的安全性完全取决于它的最薄弱环节，我们将尽最大努力使这样的薄弱环节永远不是密码系统。

要清楚地认识密码系统的另外一个理由是，一旦它们实现之后，再要更新就非常困难了。操作系统只运行在单机上，更新单机上的操作系统比较容易，而且人们也经常这样做。密码系统是在计算机网络中使用的，使得计算机之间可以互相通信，要修改网络中的通信协议是一件十分可怕的事，因此很多网络仍然使用上个世纪70年代和80年代设计的协议。我们必须知道，我们今天设计的密码系统如果得到了广泛的应用，那么它有可能要在今后的30年至50年内一直被使用。我们希望到那个时候系统的其他部分已达到了较高的安全水平。

9.1 创建正确的程序

实现的核心问题是我们不知道如何写出正确的程序或模块（所谓正确的程序就是它的运行与它的说明书完全一致），有些原因使得我们在编写正确的程序时会遇到一些困难。

9.1.1 说明书

第一个问题是大多数程序都缺少一个能清楚说明其功能的详细说明书。如果没有这样的说明书，程序正确性的概念就不能确定，从而也就无法检查程序的正确与否。

许多软件工程都有一个称为功能说明的文档，在理论上这就应该是程序的说明书，但实际上这样的文件可能根本就不存在，或者也许存在但不够完整，也可能仅仅对一些与程序运行不相关的事情进行了说明。没有清楚的说明书，就不可能写出正确的程序。

在进行程序说明的过程中，有以下三个阶段：

需求说明： 需求说明是对程序所要完成的功能的非正式描述，是回答“我用它可以做什么”的文件，而不是描述“我如何用它来做某件事”的文件。需求说明更注重对宏观结构的说明，而忽略了说明具体的细节，因此往往有些含糊。

功能说明： 功能说明对程序的执行进行详尽的说明。功能说明可以仅对那些在程序的外部可以度量的部分进行说明。

对说明书中的每一项内容，首先要自问能否对运行完的程序建立一个测试，以确定该项目成立与否。这个测试可以与内部的运行无关，而仅使用程序的外部特性。如果不能为说明书中的某一项进行这样的测试，那么，该项就应从说明书中删掉。

功能说明应该是完全的，也就是说，每一个功能都要说明，所有未在功能说明书中说明的功能一定不能完成。

从另外一个角度看，功能说明书是测试程序的根据，说明书中的任意一项都可以、也应该被测试。

实现设计： 这个文件有很多名字，它刻画了程序内部的工作方式，包括了所有不能从外部进行测试的部分。好的实现设计通常将程序分成几个模块，并对各个模块的功能进行说明。对这些模块的描述可以看做是模块的需求说明，而且要对这些模块重复这三个阶段的工作。

在这三个阶段中，功能说明无疑是最重要的一个。当程序运行完后，将根据这个文件对程序进行测试。有时，程序可以从非正式的需求说明和几个实现设计的草图的基础上产生，若没有功能说明书，就没有办法描述在程序结束后我们获得了什么。

9.1.2 测试和修正

写出正确程序的第二个问题是“测试-修正”这个几乎通用的开发方法。程序员首先写出一个程序，然后测试它是否能正确运行，如果不能，则他们就要修正程序的隐错，然后再重新测试。据我们所知，这样也并不能保证一定能得到正确的程序，但可以获得在通常情况下运行正确的程序。

1972年，Edsger Dijkstra在获图灵奖时的讲话中说，测试只能说明隐错的存在，永远不能证明隐错不存在[23]。这句话十分正确。我们非常希望能写出可以证明其正确性的程序，但遗憾的是，现在证明程序正确性方面的技术还不足以处理日常的程序设计任务，更无法顾及整个项目了。

计算机科学家不知道如何解决这个问题。也许程序正确性的证明在将来是可能的，也许我们只是需要有更全面的基础设施和方法学。现在即使没有完美的解决方案，我们也一定要用现有的工具尽最大努力去做。

关于隐错有一些简单的规则，这些规则可以在任意一本软件工程方面的书中找到。

- 如果发现了一个隐错，首先开发一个检测这个隐错的测试程序，并验证它对这个隐错的测试，然后修正这个隐错，并确保测试程序不再能检测到这个隐错。最后，让这个测试程序运行在每一个版本上，以确保这个隐错不再出现。
- 一旦发现了一个隐错，考虑引发它的原因，同时全面检查程序，看在程序的其他地方是否还有类似的隐错。
- 对发现的每一个隐错进行跟踪。对隐错进行简单的统计分析可以显示程序的哪一部分容易有隐错，或哪一类错误会经常发生，等等，这种反馈对一个质量控制系统来说是必要的。

讨论软件质量的书不少，它们的观点并不完全一致，其中许多提出了一种特殊的软件开发方法作为解决问题的手段，我们一直对这种一劳永逸的方法持怀疑态度。

9.1.3 温和的态度

第三个问题是计算机领域里的人们对程序中的错误有不可置信的温和态度，于是程序中的错误被认为是自然而然的事情。如果你的文字处理系统突然崩溃，将一天的工作全部丢失，每个人都认为这是正常的，也是可以接受的。通常他们会指责用户说：“你应该时常保存你的工作。”软件公司照样发行带有许多隐错的产品，如果他们卖的仅仅是计算机游戏软件，那倒也无所谓，而现在我们工作、经济等日常生活都依赖于软件。如果汽车制造商发现卖出去的汽车有瑕疵，他们会将这些汽车召回并进行修正。软件公司逃避了这样的责任，假设他们生产

的是其他的产品，这有时是不允许的。在客户这种温和的态度下，人们就失去了生产正确软件的强烈愿望。

9.1.4 如何着手

现实情况非常复杂，千万别以为你只需要一个好的程序员，或者是经过ISO 9001认证的开发过程、广泛的检测、甚至是这几种需要的组合就足够了。软件很复杂，不可能被这几条规则和过程所囊括。研究一下航空工业这个现实中最好的工程质量控制系统对我们有一定的指导意义。航空工业的安全系统涉及到这个行业里的每一个人，对每一步操作都有严格的规范和程序，而且在出现错误时有多重补救措施。飞机上的每一个螺母和螺钉在使用之前都必须经过飞行质量的考核认定，任何时候只要机修工带螺丝刀上飞机，他的工作将由管理员进行检查。每一次维修都有详细的记录，对任何事故都要进行谨慎的调查研究，找出引发事故的原因，然后进行修正。这种质量追求会有很高的成本，但也带来了令人瞩目的效果。今天的飞行完全是常规性操作，而每一个失误都可能是致命的。当出现问题时，飞机不可能用刹车停下来，安全返回地面的惟一途径是经过精密的操作降落在世界各地的很少的机场上。航空工业在保证飞行安全方面获得了令人瞩目的成绩，我们应尽量从中学到更多的东西。编写正确的程序的成本也许比我们现在的花费要多得多，但与现在全社会为软件中的隐错所付出的代价相比，可以肯定这样做是划算的。

9.2 创作安全的软件

到现在为止，我们只讨论了软件的正确性。仅仅要求编写正确的软件对安全系统来说是不够的，软件还必须是安全的。

二者之间有什么差别呢？正确的软件有一个特定的功能，如果按下按钮A，就会发生B。安全软件需要一个附加的要求：缺乏某功能，即不论攻击者采取什么措施，它都不可能做X。因此二者有本质上的不同，你可以检测指定的功能，但不能证明某个功能不存在。没有一个有效的方法能检验软件的安全性，这使得编写安全的软件比编写正确的软件更困难。必然的结论就是：

标准的实现技术完全不适合于编写安全的代码。

我们实际上并不知道如何编写安全的代码。软件质量涉及很广泛的领域，需要几本书来论述。对此我们掌握的并不多，但我们了解密码学及其经常遇到的问题，这也就是本章后面将要讨论的内容。

在开始讨论之前，首先声明我们自己的观点：除非你愿意为安全的实现付出努力，否则没有必要为密码学而费心。密码系统的设计很有趣，但是如果不能很好地实现，那实际上是浪费时间。

9.3 保守秘密

当你的工作涉及密码学时，就必须处理秘密信息，而且这些秘密信息还必须被保存。这就意味着要保证处理秘密的软件不能泄露这些秘密。

对于安全信道，我们有两类秘密：密钥和数据。这两类秘密都是瞬时的秘密，不需要长期存储它们。数据只有在处理每个消息的时候存储，而密钥也仅仅在安全信道期间保存。这里我们将讨论瞬时秘密的保存，关于长期秘密的保存将在第 22 章讨论。

瞬时秘密保存在计算机的内存里，很遗憾，大多数计算机的内存都很不安全。我们将依次分析每一个问题。

9.3.1 清除状态

编写安全软件的一个基本准则是立即将不再使用的数据擦除。数据保存的时间越长，其他人获取数据的机会就越大。进一步说，在你对存储介质失去控制之前，应该擦除里面存储的信息。对于瞬时秘密，这就要求清除内存的特定区域。

这样做听起来很容易，但它会引发很多问题。如果你用C语言编写了一个完整的程序，可以只关心该程序的清除。如果你编写了一个供他人使用的程序库，那么就需要依靠主程序来决定哪些状态不再需要。例如，当通信连接关闭的时候，应该通知保密库以使它清除安全信道的会话状态。这个程序库有一个函数专门完成这项工作，但我们都知道程序员可能根本不调用这个函数，因为不用这个函数程序也运行得很好。

在面向对象的语言中，问题略微简单了一些。在 C++ 中，每一个对象都有一个毁灭函数（destructor function），这个毁灭函数可以清除状态。这是 C++ 中有关安全性编码的标准实践。只要主程序合理并销毁了所有不再需要的状态，内存就将被清除。C++语言保证那些已分配堆栈的对象在处理意外事件期间堆栈被展开时一定会被摧毁，这个程序确保摧毁堆式存储的对象。调用操作系统的函数退出程序可能不必展开调用的堆栈。即使程序将要退出，也必须清除所有重要数据。操作系统毕竟不能保证立即清除这些数据，有些操作系统甚至在进行下一个应用之前不清除这些数据。

即使这些工作都完成了，我们的目的可能仍然没有实现。有些编译器很难进行优化。与安全性相关的函数用局部变量完成计算，然后清除这些变量，在C语言中可以通过调用memset函数完成。一个好的编译器会对memset函数进行更有效的直接编码优化，但其中一部分不能令人满意。它们将对检测到的不再使用的变量或数组进行清除，优化memset函数，这样做虽然很快，但程序会突然不能正常运行。代码泄露那些碰巧能在内存中找到的数据也是常有的事，如果分配给某些库的内存没有被清除的话，这个库有可能将这些数据泄露给攻击者，因此要检查编译器产生的代码，确信保密数据已被清除。

在像Java这类的语言中，事情变得更复杂。所有的对象都存储在堆栈里（无用存储单元的集合），这就意味着在无用单元收集程序找出哪个对象不再被使用之前，终结函数（finalization

function，类似于C++中的解除函数）不能被调用。对无用单元收集程序的调用频率并没有明确的说明，因此保密数据在内存中保存多长时间完全凭想像，并且异常处理的采用使得很难用手工进行清除。如果抛弃异常处理，堆栈调用会展开，此时程序设计者不能插入他自己的代码，除非将每一个函数都写成一个很大的尝试子句（try clause）。这种方法很别扭，因此很不实用。因此，异常处理必须在整个程序中都使用，这使得不可能为Java建立一个恰当而安全的库。在进行异常处理期间，Java恰当地展开堆栈，丢弃了没有被清除的对象。从这个意义上看，Java确实不好，而我们能提出的最好方法是至少保证终结例程在程序退出时运行。`main`程序使用的主要方法是采用`try-finally`语句，在finally模块中包含有强制进行无用内存收集的代码，引导无用内存收集程序完成全部终结方法（详见函数`System.gc()`和`System.runFinalization()`）。这仍然不能保证终结方法得以运行，但这已是我们能找到的最好方法了。

我们真正需要的是程序设计语言本身所提供的支持。在C++语言中，编写能立即清除不再使用的状态的程序至少在理论上是可能的，但它的其他许多特性使得它不适合于安全软件。Java使清除状态变得很困难。一个改进方法是将变量定义为敏感性的，从而保证能将它们清除。最好的解决方案是有一种能清除所有不再使用的数据的语言，这样可以避免很多错误而且不太影响效率。

保密数据可以到达其他地方，所有的数据最后都进入CPU的寄存器。清除寄存器在大多数程序设计语言中是不可能的，但对Pentium这样的CPU来说，任何数据都不可能在寄存器中存在很长时间。

在一个上下文切换期间（当操作系统从一个正在运行的程序切换到下一个程序时），寄存器中的数据将被存放在内存中，这样数据可以保存更长的时间。据我们所知，除依赖操作系统确保这些数据的保密性外，我们对此也无能为力。

9.3.2 交换文件

为了能并行执行更多的程序，多数操作系统（包括当前Windows和UNIX的所有版本）都使用虚拟内存系统。当运行一个程序时，并不是所有的数据都保存在内存里，其中有些被存在一个交换文件中，当程序要访问那些不在内存中的数据时，程序就被中断，虚拟内存系统从交换文件中读出所需要的数据并放入内存，然后这个程序继续运行。另外，当虚拟内存系统需要更多的内存空间时，它将会从被某个程序占用的空间中任取一块，并将这个程序写入交换文件。

当然，大多数虚拟内存系统没有采取任何有效措施来保护数据的保密性，在将数据写入磁盘前也不进行加密。多数软件是为友好的运行环境而设计的，而不是为密码学家所考虑的那种存在敌手的运行环境而设计的，因此自然会产生这样的问题：虚拟内存系统可以从我们的程序所占用的空间中获取内存，并将我们的程序写入交换文件，而我们的程序却不能知道这一切。假如这恰好发生在存放秘密数据的内存空间，一旦计算机崩溃或关机，这些数据仍然保存在磁盘上。大多数操作系统在正常关机的情况下，这些数据也会被保存在磁盘上。关键是并没

有清除交换文件的机制，因此这些数据就会永远地存留在磁盘上。没有人能知道交换文件将来会被什么人访问，我们确实不能承担将我们的秘密数据写入交换文件这样的风险①。

我们如何阻止虚拟内存系统将我们的数据写入磁盘呢？在有些操作系统上，我们可以使用系统调用，通知虚拟内存系统使指定的部分内存不被交换。我们几乎找不到对要进行交换的数据进行密码学意义上的安全保护的操作系统。

所有的内存空间都可以保存秘密数据，假如我们可以锁定内存以阻止其进行交换，那么应该锁定哪些内存呢？这就带来了第二个问题。在许多编程环境里，获悉数据的精确存储位置很困难，对象常常分配在堆栈里，数据可以静态分配存储空间，详细的计算很复杂，也非常容易出错。最好的解决办法也许是锁定你占用的所有内存，其实这也不像听起来那么简单，因为这样就可能失去操作系统提供的很多服务，如动态分配堆栈，而且此时虚拟内存系统不再有效。

问题的解决不应该这么困难。当然，合适的办法是创建能保护数据保密性的虚拟内存系统，这需要改变操作系统，因此不在我们的能力范围之内。即使操作系统的下一个版本有这样的特性，也要仔细检查以确认虚拟内存系统是否能保护数据的秘密性。

9.3.3 高速缓冲存储器

现代计算机不是仅有一类内存，而是具有层次结构的内存。在最底层的是主内存，大小一般为几百兆。由于主内存相对比较慢，故需要有高速缓冲存储器，它是存储量比较小但速度比较快的内存。缓冲存储器保存了内存中最近使用的数据的副本，如果CPU要访问这些数据，它首先检查缓冲存储器。如果数据在缓冲存储器中，则CPU就会以相对较快的速度得到这些数据；如果数据不在缓冲存储器中，则CPU就从主内存里读数据，并把数据的副本保存在缓冲存储器中以备将来使用。为了在缓冲存储器中让出空间，其他一些数据的副本将被丢弃。

缓冲存储器保存了一些数据的副本，其中包括秘密数据，这对安全性来说很重要。问题是当我们试图清除秘密数据时，清除操作可能并不完全。在有些系统里，对数据的修改仅仅写在缓冲存储器中，而没有写入主内存，只有当缓冲存储器需要更多的空间存储其他数据时，数据才最终写入主内存。我们并不知道这些系统的所有细节，它们会随CPU的不同而改变。我们无法知道在内存分配和缓冲存储系统之间是否存在相互影响，而缓冲存储系统可能会导致当内存在缓冲存储器被刷新前解除分配时某些清除操作忘了向主内存写入数据。制造商从来不对如何确保数据被清除的问题进行说明，至少我们没有看到这样的说明。只要没有这方面的说明，我们就不能相信它。

缓冲存储器的第二个危险是，在某些环境下，缓冲存储器知道特定的内存分配已被修改，可能是被多CPU系统中的其他CPU修改的。缓冲存储器就将为这个内存分配准备的数据作为无效数据，实际数据则没有被清除，于是没有被清除的秘密数据的副本就又一次有存在的可能。

① 事实上，我们绝不应该将未加密的秘密数据保存在任何永久性的介质上，后面我们将对此进行讨论。

对此我们几乎没有什么办法，但由于在多数系统中只有OS代码可以直接访问缓冲存储器机制，因此危险并不大。无论如何我们必须相信操作系统，于是也只有将安全性交托给操作系统了。然而，由于这些设计并没有提供实现安全系统所必需的功能，因此我们仍然很关注它们。

9.3.4 内存保留数据

在内存里简单地重写数据并不能删除数据，这令很多人感到惊奇。具体情况在一定程度上与内存的类型有关，但一般来说如果将一个数据存储到内存的某一存储单元，这个存储单元会逐渐“认识”这个数据，当进行重写或关机后，旧数据并没有完全丢失。根据具体环境，仅靠先断电再通电就可以恢复部分或全部旧数据。如果使用测试模式访问内存，其他内存可以“记住”旧数据[39]。

有几种情况可以引发这种现象。如果相同的数据存入SRAM（Static RAM）内存的同一单元一段时间，那么该数据将变成这个内存单元预置的初始状态。我们的一个朋友在很久以前使用home-built计算机时就遇到了这种现象[9]，他在写一个BIOS时在一个特殊的单元使用了一个魔值（magic value），以决定复位（reset）是冷启动还是热启动①。一段时间后机器不能通电启动，因为内存已认识了这个魔值，启动过程将每次的复位当成了热启动，这样就不对变量进行初始化了，从而使启动失败。这种情况的解决方法是交换内存芯片，让SRAM认识不规则的魔值。这个事实告诫人们：内存保存的数据比你想到的要多。

在DRAM中的情况尽管要复杂一些，但也有类似的事情发生。DRAM依靠在非常小的电容器上存储一个小电荷来工作，电容器周围的绝缘材料将会受到它的作用，导致这些材料发生变化，特别是会引起杂质迁移[39]。攻击者用物理方法可以恢复这些数据。

这是否是一个值得注意的威胁仍然有争议，但我们认为这很重要。如果你的计算机被泄露（如被偷走），你一定不想让已清除的数据也被泄露。为了达到这个目标，我们必须使计算机忘记过去的信息。

对这个问题我们只可以给出一个局限的解决方案（如果我们对内存做一个合理的假定），这个方案是有效的。该方案称为Boojum②，对相对比较少的数据量，比如密钥，它可以解决问题。设m是我们要存储的数据，替代直接存储数据m，我们生成一个随机数R，然后存储R和$R \oplus m$。这两个值被存储在内存的不同单元，最好不要靠得太近。其中的诀窍是有规律地改变R，以一定的时间间隔（比如说是100 ms）生成新的随机数R'，并将存储的数据更新为$R \oplus R'$和$R \oplus R' \oplus m$，这样可以保证写入内存里的每一比特都是随机的比特串。若要清除内存，只需用0替换m写入即可，结果是两个存储单元里存放了相同的随机数。

① 那时候home-built机用二进制形式的机器语言直接编程，这会导致很多错误，恢复已崩溃的程序的一个可靠方法是让机器复位。冷启动就是通电后启动，热启动就是使用者按复位键后运行。热启动不重新初始化所有的状态，从而并不清除使用者的设置。

② 按照Lewis Carroll的The Hunting of the Snark命名[15]。

注意，不要让R和$R \oplus m$的比特位在RAM芯片上靠得很近。由于不知道RAM芯片的工作方式，这很难做到。但多数内存将信息存储在一个比特矩阵里，用某个比特地址选择行，用另外一个选择列。如果两个部分的存储地址相差`0x5555`，则它们很有可能被存储在邻近的芯片上（这里假定内存不使用偶变址作为行号、奇变址作为列号，但我们还没有见到过这样的设计）。一个更好的方法也许是在非常大的地址空间里随机选择两个地址，这样两个单元邻近的概率就非常小，并与内存芯片的布局无关。

这仅仅是一个有局限性的解决方案，而且还很烦琐，它仅适应于较小的数据量，否则更新函数的开销会很大。但是该方案能够保证那些由秘密数据决定是否不断被重读的内存芯片并没有物理上的区别。

这种方法不能保证内存得以清除。如果你从一个内存芯片读取文件，并不能阻止内存芯片保留所有曾经存储的数据，于是，我们最大只能达到实用安全性。

在此我们只考虑了主内存，对缓冲存储器，只要能控制数据的存储位置，就可采用同样的方法解决问题。这种方法不能用于CPU的寄存器，但大量不同的数据要频繁地使用这些寄存器，我们觉得不会有数据保留问题。另一方面，扩展寄存器很少被使用，如浮点寄存器和MMX式寄存器，因此会有数据保留问题。

如果有大量的数据需要保密，这种方法要存储两个副本，同时定期地将它们与一个随机数进行异或，因此其代价将会很高。另一个较好的方法是对数据块进行加密，然后将密文存储在内存。仅有密钥需要使用这种方法（例如使用Boojum）来避免数据保留。详细内容可参阅[24]。

9.3.5 其他程序的访问

在计算机上保存秘密数据还有另外一个问题：计算机里的其他程序也可能会访问这些数据。有些操作系统允许不同的程序共享内存。如果其他程序读取了你的密钥，这会产生一个十分严重的问题。共享的内存通常必须由两个程序设立，这降低了危险。在其他情况下，共享的内存由调用共享的程序库自动设立。

现代操作系统都设计成具有被调试器使用的特性，于是调试器是一个很危险的工具。Windows的各种版本都允许将一个调试器附在正在运行的程序上，这样调试器就可以做很多事情，包括从内存中读取数据。在UNIX下，有时可以强迫对程序进行核转储（core-dump）。核转储是一个文件，它包含有程序数据的内存映像，内存映像包括所有的秘密数据。

另外一个危险来自于有特权的用户，称为超级用户或管理员。这些用户可以在计算机上得到一般用户不能得到的东西。例如，在UNIX下，超级用户可以读取任意内存区域里的数据。

一般来说，程序自身不能有效地防御这些攻击。如果你很小心，就可以排除其中一些问题，但你经常会发现你所能得到的很有限。尽管如此，仍然应该在你使用的平台上考虑这些问题。

9.3.6 数据的完整性

除了要保证数据的秘密性外，我们还要保护所存储的数据的完整性。在传输期间我们用MAC来保护数据的完整性，但如果数据可以在内存里进行修改，仍然会有问题。

在下面的讨论中，我们将假定硬件系统是可靠的，如果硬件系统不可靠，那就没有什么可做的了。如果不能保证硬件系统的可靠性，也许应该花一些时间和一定数量的内存进行验证，尽管这实际上是操作系统所做的工作。我们力争要做的是证实计算机上主内存是ECC（error-correcting code，纠错码）内存[①]，如果只有1比特发生错误，纠错码会检测到这个错误并进行修正。不用ECC内存校验，任意1比特的错误都将导致CPU读取了一个错误的数据。

为什么说这很重要呢？假设工作完成得很好，每秒钟每一比特发生错误的可能性仅为10^{-15}，如果内存的大小为128 MB，大约共有10^{12}比特，那么发生1比特错误的期望时间大约为1000秒，或者是17分钟，我们不能接受这样的出错率。出错率将会随着内存数量的增加而提高，如果内存大小为1 GB，情况就更糟。服务器通常采用ECC内存校验，因为它们的内存很大，而且运行的时间也较长，我们希望所有的计算机都有相同的稳定性。当然，这是硬件问题，你不能为运行应用程序的计算机指定内存类型。

有些问题既威胁数据的机密性，也威胁数据的完整性。

调试器有时可以修改计算机的内存数据，超级用户对它也可以直接进行修改，对此你不会有什么办法，但注意这些问题还是有益的。

9.3.7 需要做的工作

在计算机上保守秘密不像听起来那么简单，有很多渠道都可以将秘密泄露出去。你必须有效地阻止这些泄露。很遗憾，现在的操作系统和程序设计语言都不支持完全阻止这些泄露所需要的功能，你必须尽力而为，这要涉及很多工作，而且这些工作都与你的工作环境有关。

这些问题也使得创建包括密码函数的程序库非常困难。保护秘密数据的安全要涉及对主程序的修改，当然，主程序也处理保密信息，否则它就不需要密码程序库了。这是我们所熟悉的安全性与系统的每一部分有关的观点。

9.4 代码质量

如果要具体实现一个密码系统，代码质量必将花费你大量的时间。本书不是关于程序设计方面的书，但由于代码质量问题一般又被程序设计方面的书所忽略，我们下面将对此进行简单的讨论。

① 你必须证实计算机上的所有组件都支持ECC内存校验。对那些比较便宜的内存块要小心，它们不保存额外的信息，而会在空闲时重新计算这些信息，这损坏了ECC内存校验的目的。

9.4.1 简洁性

复杂性是安全性的主要敌人，因此，任何安全性设计都要力争做到简洁。在这一点上我们毫不妥协，为此我们甚至不太受欢迎。去掉所有可去掉的选项，摆脱那些很少有人使用的复杂结构。简洁在安全性中占有首要位置。

我们的安全信道是一个典型的例子。它没有选项，不允许只对消息进行加密而不认证，也不允许对未加密的消息进行认证。人们总是想获得这些性质，但他们并不知道使用这些安全性的后果。大多数用户对安全性的理解并不足以使他们有能力对安全性选项做出正确的选择，最好的方法是不要任何选项，确保默认的安全性。如果一定要有选项，就只提供单项选择：安全的或不安全的。

许多系统也有多个密码组件，用户可以选择使用加密函数或认证函数，如有可能就消除这种复杂性，选择对所有可能的应用都足够安全的单一模型。各种不同的加密模式在计算方面的差异并不大，而且对现代计算机来说密码系统也不是瓶颈。除了要摆脱复杂性之外，还要克服由用户使用弱密码组件带来的危险。如果选择加密和认证模式很困难，以至设计者都不能够正确地选择，那么有什么理由可以假定用户一定有能力做出正确的决定呢？

9.4.2 模块化

即使已经消除了大量的选项和特性，得到的系统仍然会很复杂。使这样的复杂系统易于管理的技术就是模块化。模块化就是将系统分成一些模块，然后分别设计、分析、实现每一个模块。

你可能对模块化已经很熟悉了，模块化在保证密码系统的正确性方面起到了非常重要的作用。以前我们把密码基元作为模块。模块的接口应尽量简单易懂，它的行为应与用户合理的预期一致。仔细研究模块的接口，通常会有这样一些选项或特性，它们被用于解决其他模块的问题。每一个模块都应该只解决自己的问题，因此如有可能，应该尽量删去这样的选项或特性。我们发现模块接口开始产生怪异特性时，也就是要重新设计软件的时候了，因为它们的产生总是源于设计上的不足。

模块化是我们处理复杂系统的惟一有效的方法，因此它很重要。如果一个特殊的选项只与惟一的一个模块有关，那么它就可以在这个模块的范围内分析。然而，如果选项改变了这个模块的外在行为，它就可以影响其他的模块。假如你有20个模块，每一个模块都有一个二元选项来改变模块的行为，那么就有超过百万种可能的选项组合。为了保证安全性，必须分析每一种组合，这是一个不可能完成的任务。

我们发现许多选项是为了提高效率而设计的，这是软件工程方面众所周知的问题。许多系统都进行所谓的最优化，这其实没有什么用，并不能达到预期的效果或作用很小，因为它们所优化的并不是构成系统瓶颈的那部分。对这样的最优化我们持保留态度，一般不会为此而费心。我们创造了一个精巧的设计，力争能使任务可以大批量地完成。老的IBM PC BIOS就是

一个典型的例子：在屏幕上显示一个字符的例程以单个字符为变量。这个例程的系统开销几乎花费了所有的时间，但却仅有很小的一部分用于将字符显示在屏幕上。如果例程的接口允许以字符串为变量，那么整个字符串可以用比显示单个字符所需的时间略多一点儿的时间显示在屏幕上，结果是所有的DOS机器的显示速度都很慢。所以应确保任务可以大批量地完成，而且仅仅优化那些对程序性能产生重大影响的部分。

9.4.3 声明

声明是改善代码质量的有用工具。

当编写密码系统的代码时，要有一种近乎狂热的专业态度。每个模块都不能信任其他模块，总是要检测参数的有效性，强迫执行调用顺序的限制，并且拒绝执行不安全的操作，这大多都是直接声明的。如果模块声明要求在使用一个对象前必须进行初始化，那么在初始化之前使用它就会产生一个声明错误。声明失败总是导致程序的异常中断，并带有一个说明是哪一个声明失败及失败原因的文件。

一般的规则是：在可以对系统内部的一致性进行检查时，就应该增加一项声明。捕捉一切可以捕捉到的错误，不论是你自己的还是其他程序员的。由声明发现错误并不能使程序到达安全的彼岸。

有一些程序员在研发阶段实现了声明检测，但当他们出售产品时就将这些声明检测关闭了。有谁能想到会有这样的事情呢？如果有一个核发电站，它的操作员在拥有全部安全系统的情况下进行培训，而当他们开始对实际的反应堆进行工作时，安全系统全部被关闭；或者是一个伞兵在地面训练的时候带着降落伞，而当他从飞机上跳下时却没带降落伞，那么你会怎么想呢？为什么要关闭产品代码的这些声明检测呢？这才是你惟一需要它们的地方！如果产品代码的声明失败，你会遇到一个程序错误，忽略这个错误极有可能产生错误的结果。原因就是代码的一个假设是错误的。一个程序所能做的最坏的事情就是产生错误的答案，但若至少能让用户知道出现了错误，这样的程序还可以接受，因为这使得用户可以不相信错误的结果，因此应打开所有的错误检测程序。

9.4.4 缓存溢出

缓存溢出是IT工业界的一个棘手的事情，我们将在本节讨论它。对缓存溢出问题的认识已有40年的历史，彻底解决这个问题的可用方案也具有相同的历史。一些早期的高级程序设计语言（如Algol 60）采用强制的数组边界检测就完全解决了这个问题。即使这样，Internet上一半的安全问题都是由缓存溢出引发的，而且人们仍然在拒绝用一些工具来消除它。我们认为这是一种犯罪，它好比是一个汽车制造厂用蜡纸来做汽车的油箱。当然，如果一切都正常，这也不会有什么问题，但尽管如此我们也会将它的CEO送进监狱。出于某种原因，大部分的IT企业似乎都不为其所作所为带来的后果承担责任（也许是法律制定者们使他们逃避了惩罚，这

在其他领域将是不合理的)。由于这种观点的盛行，我们有时会怀疑像密码系统这样尽力地去做某些事情是否值得。

这些都不是我们所能改变的事，但我们可以给出一些如何编写好的密码系统代码的建议。不要使用允许缓存溢出的程序设计语言，特别是不要使用C或C++；不论使用哪种语言，不要关闭对数组边界的检测。这虽然是一些很简单的规则，却可以解决一半的安全方面的隐错。

9.4.5 测试

广泛的测试总是研发过程的一部分，它可以帮助我们发现程序中的隐错，但对发现安全漏洞却没有什么用。千万不要将测试与安全性分析混淆，这两者互相补充，但完全不同。

应该进行的测试共有两类。第一类是由模块功能说明产生的一般测试集。理想的方法是让一个程序员实现模块，而让另一个程序员实现测试，二者都按照功能说明工作，如果他们之间出现任何误解，就表明必须对功能说明进行进一步的阐明。一般测试覆盖了模块的所有功能。对有些模块，这比较简单；对另外一些模块，测试程序必须模拟整个运行环境。在我们自己的很多代码中，测试代码差不多与功能代码一样长，而且我们找到了一个有效的改进办法。

第二类测试集是由模块程序员自己开发出来的测试程序，用来测试对实现程序的限制。例如，如果一个模块内部使用一个4 KB大小的缓存，对这个缓存区的起点和终点进行边界条件的附加测试将有助于发现任何缓存管理错误。这类测试的设计有时需要对模块内部有所了解。

我们常常需要一个由随机数发生器产生的测试序列，我们将在第10章对伪随机数序列(PRNG)进行广泛的讨论。使用伪随机数使得进行大量的测试变得很容易，如果我们保存了产生伪随机序列的种子，就可以重复同样的测试序列，这对测试和调试非常有用。具体细节取决于所分析的模块。

另外，我们发现设计一些在每次程序启动时都运行的快速测试很有用。Niels在最近的一个项目中要实现AES，初始化代码在几种测试情况下运行AES，用已知的正确结果检测它的输出，如果AES代码在应用程序以后的某个时刻不稳定，这个快速的测试很可能会发现这个问题。

9.5 侧信道攻击

有一类攻击我们称之为侧信道攻击(side-channel attack)[49]。当攻击者有另外一个关于这个系统的信息通道时，有可能发生这类攻击。例如，攻击者可以详细地估量系统加密一条消息所需的时间。对嵌入智能卡的密码系统，攻击者还可以估计这个卡需要的电流。磁性域、RF辐射、耗费功率与时间以及对其他数据通道的干涉都可以用于侧信道攻击。

对那些设计时没有考虑侧信道攻击的系统，侧信道攻击成功的可能性很大，对此不必感到惊奇。对智能卡进行功率分析也很容易获得成功[57]。

防御各种形式的侧信道攻击即使有可能也一定非常困难，但可以采取一些简单的预防措施。多年以前，当Niels致力于在智能卡上实现密码系统时，设计规则之一就是让CPU执行的指令序列仅仅依赖于攻击者可得到的信息。这可以防止即时攻击（timing attack），并且由于所执行的指令序列没有泄露任何信息，功率分析攻击（power analysis attack）也变得更加复杂。这种方法并不能彻底解决问题，现在的功率分析技术完全可以攻击过去的智能卡。抵抗侧信道攻击的方法将是各种对抗手段的组合，它们有些在密码系统的软件系统中，有些在硬件系统中。

防止侧信道攻击是一场速度竞赛，当你尽力去抵御已知的侧信道攻击时，而某个地方的一个聪明人又发现了另外一个侧信道攻击，于是你就必须返回去将这个攻击也考虑在内。在现实生活中，情况并没有这么糟，因为大多数侧信道攻击都难以实现。侧信道攻击对智能卡确实构成了威胁，因为智能卡完全在敌手的控制之下，而仅有很少几种侧信道攻击对其他计算机有效。实际上，最重要的侧信道攻击是即时攻击和RF辐射攻击（智能卡也容易受到功耗估计攻击）。

9.6 结论

我们希望通过本章的讨论说明这样的观点——安全性不只与密码系统的设计有关。系统的各个方面都与安全性有关，它们都各尽其责。

密码系统的实现本身是一门艺术，最关键的问题是代码质量问题。劣质的代码是现实中受到攻击的主要原因，但这也很容易避免。如果计算产品从开始到完成的时间（注意，不是开始到最初有隐错的版本的时间），按照我们的经验，编写优质代码与编写劣质代码二者所用的时间大致相同。对保证代码的质量要有一种近乎狂热的态度，代码质量可以很好，也必须很好。

最理想的是能重新设计我们的整个环境，包括我们使用的程序设计语言和操作系统，使它们都能将安全性作为基本目标之一。我们愿意致力于这项工作，如果你打算为你能确实相信的计算机花费几百万美元的话，请跟我们联系。

第二部分

密 钥 协 商

第 10 章　随机性

第 11 章　素数

第 12 章　Diffie-Hellman

第 13 章　RSA

第 14 章　密码协商协议介绍

第 15 章　密钥协商协议

第 16 章　实现问题（II）

第10章 随 机 性

为了生成密钥，我们需要使用随机数发生器（RNG）。产生好的随机性是许多密码运算中关键的一步，也是最困难的一步。

随机性有很精确的数学定义，但是由于太复杂，我们这里就不详细讨论它的实质含义了。随机性的形象化描述是这样的：即便攻击者想用各种办法来破坏随机性，随机数据仍然是不可预测的。

好的随机数发生器对于很多密码函数来说是必需的。第一部分我们讨论了安全信道以及它们的组成。我们假定有一个Alice和Bob都知道的密钥。密钥管理系统用随机数发生器来选择密钥。如果随机数发生器（RNG）有问题，那么你将得到一个易攻破的密钥。Netscape公司开发的早期版本的浏览器就是这种情况[38]。

随机性的量度叫做熵（entropy）[91]。我们不必考虑它的数学细节，但至少要给出一个直观认识。如果你有一个32比特的完全随机的字，那么它的熵是32比特。如果这32比特的字仅仅有4个值，每个值有25%的概率，那么它的熵是2比特。熵值并不是描述一个值有多少比特，而是描述这个值有多少不确定性。如果能用一个理想的压缩算法的话，我们能把熵值看做随机字的平均值。熵值的大小取决于你知道多少。一个随机的32比特的字符具有32比特的熵。现在假定你偶然知道了该字符有18比特是0，14比特是1，那么有大约$2^{28.8}$个值满足这种要求，因此熵值是28.8比特。换句话说，你知道关于字符的情况越多，它的熵值就越小。

非均匀概率分布下熵值的计算更复杂一些。对于变量X，熵值的一般定义是：

$$H(X) := -\sum_x P(X = x)\log_2 P(X = x)$$

这里，$P(X = x)$是变量X取值为x的概率。我们以后不会用这个公式，因此不需要记住它。大多数数学家谈到熵值时都会用到这个定义，而且还会用到其他几个熵值的定义。数学家们用哪个定义要看他们具体做什么工作。不要把我们提到的熵值定义与物理学家所说的熵混淆起来，他们所说的熵是用在热力学原理上的。

10.1 真随机

在理想世界里，我们采用“真随机”数据。而现实生活中，真正随机的数据是很难找到的。

一般计算机有很多熵的来源。键盘敲打的精确计时和鼠标的移动都是广为人知的例子。由于不稳定引起的硬盘访问时间的随机波动也可作为熵的来源[19]。然而所有这些来源都是值得怀疑的，因为存在着攻击者能对随机来源做测算或者能影响随机来源的情形。

很多应用者对于从各种资源中抽取熵似乎是持乐观态度的。已经有软件可以用来从单个敲击键盘的计时中产生1个或2个字节的随机数据。但我们对单个敲击键盘产生的熵的数目并不乐观。一个好的打字员能连续敲打键盘，把时间间隔控制在12毫秒之内。这种键盘敲打频率限制了计时的测量。而且即便是让用户敲击键盘来产生随机数据，我们得到的数据也不是很随机的。而且，总是存在着攻击者从"随机"事件中获取额外信息的风险。一个麦克风可以分辨出键盘的声音，这有助于确定键盘敲击的计时。我们一定要仔细地评估特定的数据包含多少熵值，毕竟我们面对的是聪明而且有主动攻击能力的敌手。

有许多随机进行的物理进程。例如，量子物理的定律要求某些行为具有完全的随机性。如果我们能测量出这些随机行为并利用它，将是很有用的。从技术上说，这一定是可行的。然而，攻击者对于这种方案有几种攻击。首先，攻击者能有意识地影响量子粒子的行为使它的发生可以预测。攻击者也能窃听我们的测量数据。如果他得到了我们测量结果的副本，虽然数据仍然是随机的，但对于攻击者来说已不存在熵值了（如果他知道了数据值，对他而言就无熵值）。也许攻击者能建立一个强RF域来试图影响探测，甚至还可能有一些基于量子物理的攻击。Einstein-Podolsky-Rosen定律能被用来破坏测量数据的随机性[5, 11]。类似的建议可应用到熵的其他来源，例如电阻的热力学噪声以及稳压二级管的燧穿和击穿噪声等。

现在一些计算机内部本身就嵌有真随机数发生器[42]。这是对分离的真随机数发生器的一个重要改进，因为它使一些攻击更困难了。随机数发生器现在仍然只能适用于操作系统，因此我们的应用程序不得不相信操作系统是以安全的方式处理随机数据的。

10.1.1 使用真随机数据的一些问题

除收集真随机数据的困难性以外，真随机数据在实际应用中还有几个其他的问题。首先，真随机数据并不总是可以得到的。如果你必须要等待敲击键盘的计时，那么如果用户没有敲击，你将不能得到任何随机数据。当你的应用是在没有键盘与之相连的Web服务器上时，这也确实是一个问题。还有一个相关的问题是，真随机数据的数量总是有限的，如果你需要很多随机数据，你就必须等待。有些事情对很多应用来说是不可接受的。

第二个问题就是，像随机数发生器这样的真随机源可以被攻破。也许用一些办法可使发生器变成可预测的。因为真随机数发生器是相当错综复杂的，他们比计算机中的其他组件更容易被攻破。如果你完全依赖真随机数发生器，当它被攻破时你的损失会很大。

第三个问题是，如何判断你能从特定的物理事件中抽取多少熵。如果随机数发生器中没有为此特殊设计的硬件，那么判定你得到了多少熵是相当困难的。我们将在后面详细讨论。

10.1.2 伪随机数据

采用真随机数据的另一个选择就是使用伪随机数据。伪随机数据并不完全是随机的，它通过确定的算法从一个种子生成数据。如果知道了种子，就能预测出伪随机数据来。传统的伪随机数发生器（PRNG）对于聪明的敌手来说是不安全的。它们被设计成统计上是随机的，但不能经受聪明敌手的攻击。Knuth所著的《计算机编程艺术》（“The Art of Computer Programming”）第二卷对随机数发生器做了详细讨论，但只是从统计随机性的角度对其进行了分析[55]。我们必须假定敌手知道用于产生随机数据的算法，给定一些伪随机的输出，对于他来说预测将来（或过去）的随机比特是可能的吗？对于很多传统的PRNG答案也许是对的，但对于严格的密码学的PRNG来说答案是否定的。

在密码系统中，我们有更加严格的要求。即使攻击者看到了很多由PRNG产生的随机数据，他都不能预测PRNG后面输出的任何数据。我们把这样的PRNG叫做密码强的。由于我们不需要传统的PRNG，我们这里仅讨论密码强的PRNG。

不要去考虑程序库中一般的随机函数，因为它几乎肯定不是密码强的PRNG。很多库里装有的PRNG，甚至连简单的统计测试都不能通过。因此如果对密码强度没有清晰的说明，不要使用库里的PRNG。

10.1.3 真随机数据和PRNG

我们仅仅使用真随机数据做简单的事情：作为伪随机数生成器PRNG的种子。这种构造解决了使用真随机数据的一些问题。一旦PRNG有了种子，就可以生成随机数据了。你可以不断地把得到的真随机数据加为PRNG的种子，这样就保证了即使以前的种子被攻破，随机数据也是完全不可预测的。

有一些学者认为真随机数据比从PRNG得来的伪随机数据要好。在密码协议中如果使用真随机数据，那么可证明攻击是不可行的，协议是无条件的安全的。如果用PRNG，协议只有在攻击者不能攻破PRNG的情况下是安全的，因此协议在计算上是安全的。但是，这种区别仅仅是像牙塔中的人们才会考虑的。所有的密码协议对每样事物几乎都使用计算上的假设。不考虑某一特定类型攻击的计算假定，只是通过无关紧要的改进而产生真随机数据来满足你所需要的无条件的安全是相当困难的。因此你可能更愿意降低系统安全，尽量使用真随机数据。真随机数发生器中的任何薄弱点都会立刻导致系统不安全。然而，如果你用真随机数据来作为PRNG的种子，关于熵源的假定就会更保守一些，你似乎更希望最终产生一个安全的系统。

10.2 PRNG的攻击模型

关于伪随机数发生器PRNG的研究并不多。从一个种子产生（伪）随机数的任务是相当简单的，问题是怎样得到随机种子以及怎样在真实世界中保护它的私密性[48]。到目前为止，我

们所知道的最好的设计叫做Yarrow[47]，这是几年前我们与John Kelsey一起设计的。Yarrow试图防止所有已经知道的攻击。

无论何时PRNG都有一个内部的状态。所需的随机数据是通过用密码算法来产生伪随机数据完成的。这个算法也更新了PRNG内部的状态，用以保证下一次请求不会产生同样的随机数据。这个过程很容易，任何 Hash 函数或分组密码都可用在这一步。

对于 PRNG 有各种形式的攻击。一种很直接的攻击是攻击者试图从输出中重新构造出 PRNG 的内部状态。这是一种典型的密码攻击，很容易通过密码技术来防范。

如果攻击者能在一定程度上获得PRNG的内部状态，防范就变得更困难了。我们的讨论并不是要探究这种攻击是如何发生的。也许在应用中有一个缺陷；也许计算机是第一次启动，还没有随机的种子；或者也许攻击者设法从硬盘上读到了种子文件。事实是糟糕的事情确实发生了，你必须能够处理它。在传统的PRNG中，如果攻击者获得了内部状态，那么他就能得出所有的输出和所有内部状态的更新。这意味着如果PRNG被成功地攻击过，那么它就再也不能恢复到安全的状态了。

要想恢复一个内部状态已经被攻破的PRNG是很困难的，我们需要从真随机数发生器中产生一些熵源。为了便于讨论，我们假定有一个或多个源（例如我们称之为事件的小块）不定时地来提供一定数目的熵。

即便我们把从事件中获得的小数量的熵混合进内部状态，仍然存在着被攻击的风险。攻击者只是正常地请求PRNG的随机数据，只要在两次这样的请求之间所加的熵的整体数量被限制在 30 比特，那么攻击者就能简单地尝试随机输入的所有可能性，并恢复混合以后新的内部状态。这将大约需要 2^{30} 次尝试，是可以做到的[①]。攻击者可通过PRNG产生的随机数据来检验是否碰上了正确的解。

对付这种特定攻击的最好防御办法是把包含熵的事件混到一起，不断地把足够多的熵收集到内部状态中，直到攻击者不能猜出混合的数据为止。那么多少是足够呢？好，如果我们想要攻击者对任何攻击都至少用 2^{128} 步，那我们就要有 128 比特的熵。但是确实存在这样一个问题：对熵的数量做任何的评估都是相当困难的。这主要取决于攻击者知道多少或能知道多少，但是这种信息对于开发者在设计阶段来说是不可能获得的。这就是 Yarrow 要解决的主要问题。它试图用熵评估器来测量源的熵值，但是这样的评估器是不可能适应所有情况的。

10.3 Fortuna

写这一章的时候，我们提出了 Yarrow 的一种改进方案，称为 Fortuna，是罗马命运女神的意思[②]。Fortuna 不使用熵评估器，从而解决了不好定义它的问题。本章的余下部分主要来讨论 Fortuna 的细节。

① 我们这里不过多地考虑数学问题。在这个例子里我们用猜测熵而不是标准的Shannon熵。有关熵测量的细节，参阅[14]。

② 我们原来想叫它 Tyche ——希腊命运女神的意思，但没有人知道怎样拼写。

Fortuna 有三部分。发生器采用固定大小的种子来生成任意数量的伪随机数据。累加器收集和混合从各种源产生的熵，有时还给发生器新的种子。最后，种子文件管理能保证即便是机器刚刚启动，PRNG 也能产生随机数据。

10.4 发生器

发生器的功能是把固定大小的状态转换成任意长的输出。我们把像 AES 这样的分组算法用在发生器中，可以任意地选择 AES（Rijndael）、Serpent 或 Twofish 算法（见 4.5.7 节）。发生器的内部固有状态包含 256 比特的分组密钥和 128 比特的计数器。

发生器基本上说是在计数器模式下的分组加密。CTR 模式产生一个随机数据流，这就是它的输出。下面是一些提炼出的主要功能。

如果一个用户或是应用请求随机数据，发生器运行它的算法来产生伪随机数据。现在假定请求完成后攻击者设法获得了发生器的内部状态。我们所希望的是，这不会导致发生器以前产生的结果被攻破。因此，当每次请求后我们产生一额外的 256 比特伪随机数据，用它作为分组加密的新密钥。我们忘记旧的密钥，因而就排除了泄露旧的请求信息的任何可能性。

为了保证产生的数据在统计上是随机的，不能一次生成太多的数据。毕竟，在完全随机的数据中有重复的分组值，而计数模式的输出从来不包含重复的分组值（见 5.8.2 节）。对于这种问题有各种各样的解决办法。我们可以仅仅用每个密文分组的一半，这样能隐藏大多数的统计偏离。我们也可以使用叫做伪随机函数的算法，而不是用分组加密，但是就我们所知还没有一个经过详细分析的有效的建议。最简单的解决办法是限制单次请求中随机数据的字节数，这样使得统计偏离更难以检测。

如果要从单个密钥中产生 2^{64} 组输出，分组值应该差不多有一个碰撞。这样几次重复请求后就会显示输出不是完全随机的，它缺少应有的分组碰撞。我们把单次请求的分组最大限制到 2^{16} 组（也就是 2^{20} 字节）。对于一个理想的随机数发生器来说，在 2^{16} 个输出分组中找到一个分组值碰撞的概率是 2^{-97}，因此直到做大约 2^{97} 次请求时，碰撞是否完全消失才能被检测到。对于攻击者来说整个过程需要 2^{113} 步来完成。虽然不是我们想要的 2^{128} 步，但已经很接近了。

我们在这里放宽了条件，接受一个（稍微）减弱了的安全度，但此外似乎没有更好的选择。我们没有任何合适的密码算法来给 PRNG 一个完整的 128 比特的安全度。我们可以用 SHA-256，但那样会更慢。人们总会无休止地争论是否用好的密码 PRNG，速度一直是争论的焦点之一。很少有人会为了多得到几个比特的安全度而减慢 PRNG。多数人仅仅是转而使用一个完全不好的 PRNG，因此整个系统的安全水平将会下降。

如果我们已经有 256 比特长的分组密文，那么碰撞就完全不成问题了。这种特定的攻击不能构成很大威胁。不仅攻击者要完成 2^{113} 步，被攻击的计算机也不得不完成 2^{113} 次分组密文加密，因此这种攻击依靠的是用户计算机的速度，而不是攻击者计算机的速度。大多数用户不会

增加大量的额外计算能力来帮助攻击者。我们不喜欢这些类型的安全措施，它们更加复杂。如果PRNG曾被用在一个特殊的环境中，那么这种方案也许在其他环境中就不能再应用了。不管怎样，对于给定的情况来说，我们的解决方案是我们所知道的最好的方案。

在每次请求完成后给分组密码重新生成密钥时，计数器不被重设。这是一件小事，但它避免了短循环带来的问题。假定我们每次都重设计数器。如果密钥值重复，而且所有的请求都是固定大小的，那么下一个密钥也将是一个重复的。这就会以一个短的密钥循环而结束。这不是我们想要的，通过不重设计数器就可以完全避免它。由于计数器是128比特，我们从不重复计数值（2^{128}个分组是在我们计算机的计算能力之外），这就自然而然地打破了任何循环。再进一步，我们用计数器值0来表示发生器还没有密钥，因此不能产生任何输出。

注意，每次请求至多有1 MB数据的限制不是不可改变的。如果需要多于1 MB的随机数据，做重复的请求就可以了。事实上，具体应用时能够提供一个界面来自动地完成这些重复的请求。

发生器本身就是一个相当有用的模型。我们期望大多数应用能把它作为界面的一部分，而不仅仅是Fortuna的一个组件。运行做Monte Carlo模拟[①]的程序。你当然希望模拟是随机的，但如果仅仅为了调试和验证的目的，也希望能重复完全同样的计算。一个好的解决办法是在程序开始的时候调用一次操作系统的随机发生器，得到一个随机种子。这个种子能被记录作为模拟器输出的一部分，我们的发生器能从这个种子产生模拟所需的所有随机数据。知道了发生器的原始种子，可以通过用同样的输入数据和种子再一次运行程序来验证所有的计算。对于调试来说，同样的模拟可以一遍又一遍地运行，只要开始的种子保持不变，每次将会完全一样地运行。

我们下面详细地说明发生器的操作。

10.4.1 初始化

这一步是相当简单的。我们把密钥和计数器设为0来表示发生器还没有种子。

function InitializeGenerator
output: g 发生器状态。
　设密钥 K 和计数器 C 为零。
　$(K, C) \leftarrow (0, 0)$
　把值传给状态。
　$g \leftarrow (K, C)$
return g

① Monte Carlo 模拟是被随机选择所驱动的模拟。

10.4.2 重新产生种子

重新产生种子的操作用一任意输入串更新发生器状态。这里我们不考虑输入串包含什么。为了保证现有的密钥和输入的完全混合，我们使用 Hash 函数。

function Reseed
input: g 发生器状态。通过这个函数状态被改变。
　　s 新的或要加入的种子。
　用 Hash 函数计算新的密钥。
　$K \leftarrow \mathrm{SHA}_d\text{-}256(K \| s)$
　计数器增加使其非零，表示发生器已有种子了。整个发生器中，C 是一个 16 字节的值，用最低字节优先规则把它看成是一个整数。
　$C \leftarrow C + 1$

这里计数器 C 被用做整数。后面它被用做一个明文分组。为了在这两者之间进行转换，我们使用最低字节优先规则。明文块是一组 16 字节 $p_0, \cdots, p_{15}$ 的明文，对应于整数值

$$\sum_{i=0}^{15} p_i 2^{8i}$$

通过使用这个规则，我们能把 C 既看做一个 16 字节串，又看做一个整数。

10.4.3 生成分组

这个函数生成一些随机输出的分组。这是一个只有发生器可以使用的内部函数。PRNG 外部的任何实体都不能调用这个函数。

function GenerateBlocks
input: g 发生器状态。通过这个函数状态被改变。
　　k 要产生的分组数目。
output: r $16k$ 字节的伪随机字符串。
　assert $C \neq 0$
　以一个空串开始。
　$r \leftarrow \varepsilon$
　加入所需的分组。
　for $i = 1, \cdots, k$ **do**
　　$r \leftarrow r \| E(K, C)$
　　$C \leftarrow C + 1$
od
return r

当然，函数 $E(K, C)$是密钥 K 和明文 C 的分组密码加密函数。函数 GenerateBlocks 首先检查 C 是否非零，因为零表明发生器还没有种子。循环以 r 中一个空字符串开始，把每次新计算的分组加入到 r 中而构成输出值。

10.4.4 生成随机数据

这个函数在发生器用户的请求下产生随机数据。它允许输出最多可达到 2^{20} 字节，保证发生器忘掉产生结果的任何信息。

function PseudoRandomData
input: g　发生器状态。通过这个函数状态被改变。
　　　n　要产生的随机数据的字节数。
output: r　字节的伪随机串。
　限制输出长度以减少与完全随机输出的统计偏差。也要保证长度非负。
　assert $0 \leqslant n \leqslant 2^{20}$
　计算输出。
　$r \leftarrow$ *first-n-bytes*(GenerateBlocks(g, $\lceil n/16 \rceil$))
　产生一个新密钥以避免这个输出以后的泄露。
　$K \leftarrow$ GenerateBlocks(g, 2)
　return r

输出是通过调用 GenerateBlocks 得到的，仅有的变化是结果被删减成正确的字节数。($\lceil \cdot \rceil$ 是取上界的算子。) 然后，我们产生两个分组得到一个新密钥。一旦旧密钥 K 已经被忘记，就没有办法重新计算结果 r 了。只要 PseudoRandomData 没有保存 r 的副本，没有忘记删去内存中的 r，那么函数一旦完成，发生器没有办法泄露关于 r 的任何数据。这就是发生器被攻击后不会影响以前输出秘密的确切原因。但它确实对将来输出的秘密有威胁，这个问题在累加器中将会考虑。

函数 PseudoRandomData 限制了它返回数据的数目。我们不去费力地说明通过重复调用 PseudoRandomData可以返回更大的随机串。要注意在每次调用时不应该增加最大输出数目，因为这样就增加了与完全随机的统计的偏差。重复地调用 PseudoRandomData 是很有效的。真正的负担是每产生 1 MB 的随机数据，将不得不产生 32 个额外的随机字节（新的密钥），并且还要再运行一次分组密码的密钥表。这种负担对于我们所建议使用的分组密码来说是无关紧要的。

10.4.5 发生器速度

我们阐述的 Fortuna 的发生器是一个密码强的 PRNG，它把种子转换成任意长的伪随机输出。它的运行差不多和分组密码一样快。对于 PC 机类型的 CPU，大的请求中每产生 1 个字节的运行时间少于 20 个时钟周期。它能替代大多数的 PRNG 库函数。

10.5 累加器

累加器从各种资源中收集真随机数据，用它来重新产生发生器的种子。

10.5.1 熵的源

我们假定周围环境中有几种熵的来源，每一个源都能及时地产生含有熵的事件。用什么来作为熵的源并不重要，只要至少有一个源能产生攻击者不可预测的数据即可。由于你不能知道攻击者将怎样攻击，最好的赌注就是把任何看起来像不可预测数据的事件转换成一个随机源。敲击键盘和鼠标移动都能成为理想的源。另外，你应该把实际存在的很多时间源加入。你可以采用敲击键盘、鼠标移动的准确时间以及磁盘驱动和打印机的反应时间，而且最好能同时使用。这样一来，只要攻击者不能预测或复制所有源的数据，就不能构成威胁了。

处理这些源需要大量的工作。这些源一般都被放在操作系统的各种硬盘驱动器中。对一般用户来说，应用它们几乎是不可能的。

我们利用0到255之间的惟一的源号码来标识每一个源。应用者可以选择是静态地还是动态地分配源号码。每一个事件产生的数据是一个短的字节序列。源应该只包含每一事件中不可预知的数据。例如，时间信息能用一个准确计时器的两个或四个最少的主要字节来表示，没有必要包含日、月、年。假定攻击者知道那些信息会更安全一些。

我们要把来自不同源的各种事件串联到一起。为了保证这样联起来的串能惟一地表示事件，我们要让这些字符串是可分析的。每一事件用3个或更多的字节来编码。第一个字节包含随机源的编号，第二个字节包含数据中附加字节的数目，其后的字节表示源提供的数据。

当然，攻击者会知道一些源所产生的事件。为了模拟这种情况，我们假定有一些源完全在攻击者的控制之中。攻击者可以控制这些源在指定时间产生指定事件。而且，攻击者可以像其他用户一样，随时从 PRNG 中请求随机数据。

10.5.2 混合池

为了给发生器重新产生种子，我们需要在一个足够大的池子中混合事件，使得攻击者不再能够估算出池子中事件的可能值。在“足够大”的池子中，随机事件重新产生的种子破坏了攻击者已经知道的关于发生器状态的信息。但遗憾的是，在使用前我们不知道需要在池子中收集多少事件合适。这也是Yarrow用熵评估器和各种启发式的规则来努力解决的问题。Fortuna以一种更好的方式解决了它。

假设有 32 个池子：$P_0, P_1, \cdots, P_{31}$，每个池子包含一串没有限定长度的字节。实际上，这个字节串仅仅被用做Hash 函数的输入。应用装置不需要储存这个无界的字符串，而是计算池子中不断增长的字符串的Hash 值。

每个源以循环的方式在池子中分布它的随机事件，这保证了每个源产生的熵均匀地分布在各个池子中。每一个随机事件被加到所指定池子的字符串中。

每次当池子P_0足够长时给发生器重新产生种子，这些种子被编号为1，2，3，…。根据编号r，一个或更多的池子被用于重新产生种子。当2^i被r整除时，我们用池P_i来重新产生种子r。这样一来，P_0在每次重新生成种子时都被用到，P_1每隔一次被使用一次，P_2每四次被使用一次，等等。当一个池子用于重新生成种子时，它被重新设为空。

这个系统自动地适应周围环境。如果攻击者几乎很少知道随机源，他在下一次重新生成种子时就不能预测P_0。但是攻击者也许知道很多随机源的情况，或者他也许（伪造地）产生了很多事件。在这种情形下，他可能知道足够多的P_0，能够从旧的发生器状态和发生器输出中重新构造新的发生器状态。但是当P_1同时被使用时，它就包含了两倍的攻击者不可预测的数据；P_2则是4倍。不用考虑攻击者产生了多少伪造的随机事件，或是他知道多少事件，只要至少有一个源的随机事件是他不能预测的，就总会有一个池子能收集足够多的熵来击败攻击者。

系统从被攻破的状态恢复起来的速度取决于熵（对于攻击者来说）流进池子中的速率。如果假定一个固定的速率ρ，则t秒后我们总共有ρt比特的熵。每个池子在这个时间段收到大约$\rho t/32$比特的熵。如果发生器用一个装有多于128比特熵的池子重新生成种子，攻击者就不再能追踪到它的状态了。有两种情形：如果P_0在下一次重新生成种子的操作之前已经收集了128比特的熵，那么系统就从被攻破的状态恢复起来了。这种恢复有多快取决于在重新生成种子之前我们让P_0增长了多少。第二种情形是由于攻击者知道的（或本身产生的）随机事件太多，P_0重新生成种子太快。令t表示两次重新生成种子之间的时间，则池子P_i在这段时间收集了$2^i\rho t/32$比特的熵，并且每隔$2^i t$秒被用在一次重新生成种子中。我们第一次用池子P_i重新生成种子时，系统的状态就恢复了，这里$128 \leq 2^i\rho t/32 < 256$。（如果上界不是256，池子$P_{i-1}$在这段时间将包含128比特的熵。）这个不等式

$$\frac{2^i \rho t}{32} < 256$$

也就是

$$2^i t < \frac{8192}{\rho}$$

换句话说，系统恢复的时间（$2^i t$）是被收集2^{13}比特熵的时间（$8192/\rho$）所界定的。数字2^{13}似乎有些大，但它可以用下面的方式来解释。我们至少需要2^7比特才能恢复系统。也许遗憾的是在一个特定的池子还没收集到2^7比特熵之前，系统就重新生成种子了，那么我们就不得不用下一个池子，它已经收集了接近2^8比特的熵。最终，我们把数据分在32个池子中，这相当于另一个分解因子2^5。

这是一个很好的结果。它是在一个理想结果的64倍之内（它需要至多64倍的理想结果所需的随机性）。这是一个常数因子，它保证了我们永远不会做得太糟，系统最终总会恢复。而且，我们不需要知道我们的事件有多少熵或攻击者知道多少熵。这就是Fortuna胜过Yarrow的绝对优势。现在，我们永远不再需要根本不可能构造的熵评估器。处理每一件事情都是完全自

动的，如果有一个好的随机数据流，PRNG将会很快恢复。若仅仅有很少的随机数据，就会用比较长的时间来恢复。

我们现在忽略了这样的一个事实，我们仅有32个池子，甚至池子P_{31}在两次重新生成种子之间都没有收集到足够的随机性来恢复系统。这种情况可能是这样发生的，攻击者放入了太多的随机事件，以至于随机源产生2^{13}比特熵之前已经进行了2^{32}次重新生成种子。这是很不幸的，为了阻止攻击者进行这样的尝试，我们要限制重新生成种子的速度。前一次的重新生成种子完成100毫秒后才能进行下一次的重新生成种子。这就把重新生成种子的频率限制到每秒进行10次，因此至少需要13年才能使用P_{32}。从经济方面和技术方面考虑，大多数计算机设备的寿命少于10年，故我们把池子限制为32个似乎是一个很合理的解决办法。

10.5.3 应用考虑

在累加器的设计中有几个应用方面的考虑。

事件在池子中的分布

收集来的事件应该被分布在池子中。最简单的解决办法是由累加器完成这个任务。然而，这是很危险的，因为需要有一些函数调用把事件传递给累加器。攻击者很可能也会对这个函数做任意的调用。每当一个“真”的事件生成时，攻击者就能对这个函数做额外的调用，因此影响了下一个“真”事件要去的池子。如果攻击者设法让所有的“真”事件进入池子P_0，那么整个多池系统就无效了，单个池的攻击成功。如果攻击者让所有的“真”事件都进入P_{31}，则它们肯定从没被使用过。

我们的解决办法是通过每个事件发生器把事件传递到正确的池子中。这就要求攻击者若想影响池子的选择，必须进入产生事件的程序内存中。但如果攻击者真可以进入那里，那么整个源也就可能被攻破了。

累加器能够检查每个源是否把事件放入了正确的池子中。对于一个函数来说，检查它的输入是否正确是一个好主意，因此从理论上说这是不错的。但是在这种情况下，如果验证失败了，累加器应该做什么却不是很清晰。如果整个的PRNG作为一个用户进程来运行，PRNG就会抛出一个致命错误并且退出程序，因此将仅仅因为一个源的错误而破坏了整个PRNG系统。如果PRNG是操作系统内核的一部分，那就更困难了。我们假定由一个特定的驱动源来产生随机事件，但是此驱动源不能追踪一个简单的5比特的循环计数器。累加器应该做什么呢？返回一个错误代码？这有可能是犯这样低级错误的程序员没有检查返回代码。累加器终止内核？这又太过火了，仅仅因为一个错误的驱动源而影响了整个机器。我们所能想到的最好的办法就是在CPU时间内来延迟驱动。如果验证失败了，累加器能把有问题的驱动延迟1秒左右。

这个办法也不是很有效，因为我们让每个事件发生器来选择池子的原因就是假定攻击者也许会用假的事件来对累加器进行调用。如果发生这种情况，累加器检查池子排序后，真的事件发生器就会为攻击者的错误行为而被惩罚。我们的结论是：累加器不应该检查池子的排序，

因为如果检查出错误，它也不能够采取有效的措施。每个随机源应该以循环的方式把事件分布在池子中。如果一个随机源出现错误，我们也许会失去它所产生的熵，但不会有其他的危害。

事件传递的运行时间

我们想要限制事件传递给累加器时所必需的计算量。许多事件是计时事件，它们由实时驱动所产生。如果这些驱动调用累加器需要花很长时间的话，它们就不会进行调用。

我们需要进行一个最小量的计算。我们一定要把事件数据加到选定的池子中。当然，我们不用把整个池子中的字符串都存在内存中，因为一个池子中字符串的长度是无界的。每一个池子有一个小的缓冲区，只要缓冲区满了就计算部分Hash值。这就是每个事件所需的最小量的计算。

我们不想完成整个重新产生种子的操作，用一个或多个池子给生成器重新产生种子。这个工作将会花费大量的时间，比仅把事件加入池子中的时间多出很多。相反，这个工作要等到下一个用户请求随机数据时才去做，它会在随机数据产生之前被完成。这就把一部分的计算负担从事件发生器转移给随机数据的用户了。用户不仅不那么匆忙了，而且他们也是从PRNG服务中受益的人。毕竟，大多数事件发生器是不会从它们所产生的随机数据中受益的。

为了允许仅仅在随机数据的请求进行之前才完成重新产生种子的操作，我们必须要封装发生器。换句话说，发生器要被隐藏使得它不能直接被调用。累加器会提供一个与PseudoRandomData具有相同接口的RandomData函数。这就避免了有些用户直接调用发生器而略过我们花费很多工夫做的重新产生种子进程。当然，用户仍然可以产生自己使用的发生器实例。

一个典型的像SHA_d-256这样的Hash函数处理固定大小分组的输入。如果每组池子中的字符串一完成就被处理，那么每个事件将至多导致一个Hash分组计算。然而，这也存在不足。目前计算机用一系列缓存来使CPU保持繁忙。缓存的效果之一就是能有效地让CPU一段时间内一直做同一件事情。如果你处理一个Hash代码分组，那么CPU必须在它运行之前就把Hash函数代码读进最快的缓存中。如果你顺序地处理几个分组，那么第一个分组把代码放进最快的缓存，以后的分组也采用这种方法。总体来说，让CPU一直在一个小循环中工作，而不让它在不同的代码段中切换，这大大提高了CPU的性能。

考虑到上面的情况，我们的选择是增加每个池子缓冲区的大小，计算Hash值之前在每个缓冲区中收集更多的数据。这样的优势是整体所需的CPU时间减少了，但不足的是把一个新事件加入池子中所需的最长时间增大了。这就是一个在这里不能解决的应用矛盾。具体采用哪种方式主要依具体的环境而定。

10.5.4　初始化

初始化总是一个简单的函数。到目前为止，我们仅仅谈论了发生器和累加器，但我们要定义的函数是Fortuna外部界面的一部分。它们的名字反映了它们是在整个PRNG上运行的事实。

```
function InitializePRNG
output: R    PRNG 的状态。
  令 32 个池子为空串。
  for i = 0, ···, 31 do
      P_i ← ε
  od
  令重新产生种子的计数器为零。
  ReseedCnt ← 0
  初始化发生器。
  g ← InitializeGenerator()
  给状态打包。
  R ← (g, ReseedCnt, P_0, ···, P_31)
  return R
```

10.5.5 产生随机数据

这不是一个关于PRNG中发生器组件的简单的包，因为我们在这里必须要处理重新产生种子。

```
function RandomData
input:  R    PRNG 状态；通过这个函数状态被改变。
        n    要产生的随机数据的字节数。
output: r    伪随机字节串。
  if length(P_0) ≥ MinPoolSize ∧ last reseed >100 ms ago then
  我们需要重新产生种子。
  ReseedCnt ← ReseedCnt + 1
  加入所有我们要用的池子的 Hash 值。
  s ← ε
  for i ∈ 0, ···, 31 do
      if 2^i | ReseedCnt then
          s ← s || SHA_d-256(P_i)
          P_i ← ε
      fi
    od
    得到数据，现在进行重新产生种子。
    Reseed(g, s)
  fi
```

（如果需要）重新产生种子。让是 R 的一部分的发生器来做这个工作。

return PseudoRandomData(g, n)

这个函数开始是通过检查池子P_0的长度是否大于参数MinPoolSize来决定是否重新生成种子的。对于池子含有128比特的熵之前它的大小应该是多少，我们可以进行优化的评估。假定每个事件包含8比特的熵，把4个字节放在池子中（这相应于2字节的事件数据），那么MinPoolSize的合适值应该是64字节。这并不是很重要，虽然选择一个小于32的值似乎是不可取的，但选择一个更大的值也不合适，因为即便有很好的随机源，这样也会延迟重新生成种子。

下一步是增加重新生成种子的计数。计数值初始化为零，因此第一次重新生成种子时值为1。这自动地保证了第一次重新生成种子仅仅使用池子P_0，这也正是我们所希望的。

循环把池子的Hash值加入到种子。我们本可以把池子本身的字符串加入，但是那样的话每个应用就不得不储存整个池子的字符串，而不是仅仅运行每个池的Hash计算。其中，$2^i \mid$ReseedCnt表示一个分解测试。如果2^i是ReseedCnt的因子，值为真。聪明的读者会注意到，一旦i在这次测试中失败，循环中后面的所有测试也将失败，这是一种优化。

10.5.6 加入事件

当随机源有另一个随机事件时，就调用这个函数。应该注意的是每个随机源通过一个源编号来惟一确定。我们不去说明怎样分配源编号，因为这要根据特定的情况来决定。

function AddRandomEvent

input: R PRNG状态；通过这个函数状态被改变。

s 源编号范围0, …, 255。

i 池子编号范围0, …, 31。每个源必须以循环的方式把事件分布在所有的池子中。

e 事件数据。字节串，长度范围1, …, 32。

首先检查参数。

assert $1 \leqslant length(e) \leqslant 32 \wedge 0 \leqslant s \leqslant 255 \wedge 0 \leqslant i \leqslant 31$

把事件加入池子中。

$P_i \leftarrow P_i \parallel s \parallel length(e) \parallel e$

事件用$2 + length(e)$个字节来编码，其中s和$length(e)$用单个字节编码。这个连接被加入到池子中。要注意的是，我们仅仅说把数据加入到池子中，而没有提到任何Hash计算。我们仅在使用它的地方才会说明池子的Hash计算。一个实际的应用是应该计算Hash值的应用。虽然应用比较容易，但说明它会比较复杂。

我们已经把事件数据的长度限制到32个字节。太大的事件是没有用的，随机源不应传递大量的数据，而是应传递那些包含不可预测随机数据的小部分字节。如果一个源中包含一些熵的大量数据，它首先应对这些数据做Hash计算。函数AddRandomEvent应该总是很快地返回

值。这很重要，因为许多源本身在做实时服务。这些源不能花费大量的时间来调用AddRandomEvent。即便一个源产生了小事件，它也不必等待那些大事件的调用。大多数应用通过互斥的办法来对 AddRandomEvent 的调用顺序化，以保证在同一时间仅有一个事件被加入[①]。

一些随机源也许没有时间调用 AddRandomEvent。在这种情况下，我们就要把事件存储在缓冲区中，有一个独立的进程把事件从缓冲区中取出并传递给累加器。

另一种设计是允许源只是把事件传递给累加器进程，在累加器中有一个独立的线程来完成所有的 Hash 计算。这是一种更复杂的设计，但它对于熵源来说确实有优势。设计时如何选择主要根据实际的情况决定。

10.6 种子文件管理

到目前为止，我们的PRNG就是收集熵，并且在第一次重新生成种子后产生随机数据。然而，如果重新启动机器后，我们就必须等待随机源在得到任何随机数据之前产生足够的熵来发起第一次重新生成种子的操作。而且，我们也不能保证第一次重新生成种子后的状态对攻击者来说是不可预测的。

解决办法就是利用一个种子文件。PRNG保存一个充满了熵的独立文件，叫种子文件。这个种子对任何其他人来说是不可得到的。当重新启动一次之后，PRNG读种子文件，并用它作为熵以达到一个不可知的状态。当然，一旦种子文件以这种方式使用过，它就需要用新的数据重写。

我们将阐述种子文件的管理，首先假定文件系统支持自动的操作，后面我们将讨论在实际系统中如何应用种子文件管理。

10.6.1 写种子文件

我们要做的第一件事是产生一个种子文件，这可以用一个简单函数来完成。

function WriteSeedFile
input: R PRNG 状态。通过这个函数状态被改变。
f 要写入的文件。
write(f, RandomData(R, 64))

这个函数简单地产生了64字节的随机数据并把它写入文件。这里的数据比我们一般需要的稍微多了一些，但我们没有理由对字节过于吝啬。

10.6.2 更新种子文件

显然，我们也需要能读一个种子文件。由于下面将要讲到的原因，我们总是在同一个操作中更新种子文件。

① 在多线程环境中，总要保证不同的线程彼此之间互不干扰。

```
function UpdateSeedFile
input:  R    PRNG 状态。通过这个函数，状态被改变。
        f    要更新的文件。
  s ← read(f)
assert length(s) = 64
Reseed(g, s)
write(f, RandomData(R, 64))
```

这个函数读种子文件，检查它的长度，给发生器重新生成种子，然后用新的随机数据重写种子文件。

这个函数必须保证在重新生成种子和将新数据写入种子文件之间PRNG不能进行其他的操作。有这样一个问题：当机器重启之后，用此函数读种子文件，数据用于重新生成种子。假定攻击者在种子文件更新之前请求随机数据。只要一返回随机数据，在种子文件更新之前，攻击者就会重启机器。在下一次重启时，同样的种子文件会被读到，并用它来给发生器重新产生种子。这时一个无辜的用户在种子文件被重写之前请求随机数据，他将会得到与攻击者一样的随机数据。这触犯了随机数据的私密性。由于我们通常用随机数据来生成密钥，因此这是一个相当严重的问题。

实现应该保证种子文件是秘密的，而且，种子文件的所有更新一定是自动的（见10.6.5节）。

10.6.3 何时读写种子文件

当重启计算机时，PRNG没有熵来产生随机数据，这就是种子文件存在的原因。因此，每次计算机重启后，都要读种子文件，然后更新它。

当计算机运行时，它收集了来自各种源的熵。我们最终也想让熵来影响种子文件。一种很直接的方法是，当机器关闭的时候重写种子文件。由于有一些机器从不会以正常的方式来关闭，PRNG也应该在固定的时间间隔内重写种子文件。我们这里就不讨论细节了，因为它们常常与具体平台有关。我们要保证在PRNG收集了很多熵以后种子文件被规则地更新，这是很重要的。一个合理的方法是，在每次关闭机器时和每隔10分钟左右重写种子文件。

10.6.4 备份

重新生成种子时也会产生很多问题。我们不允许PRNG重复两次同样的状态。我们用文件系统存储种子文件来防止这种情况发生。但是大多数文件系统在设计时没有考虑避免重复同样的状态这一问题，这就引起了很多麻烦。

首先，存在备份。如果做了整个文件系统的备份后重启计算机，那么PRNG会从种子文件中重新生成种子。如果后来从备份中恢复了整个文件系统并且重启机器，则PRNG将从同样的种子文件中重新生成种子。换句话说，在累加器收集足够多的熵之前，PRNG在两次重启之后

产生同样的输出。这是一个很严重的问题，因为攻击者可以找回另一用户从PRNG得到的随机数据。

对于这种攻击，目前还不存在直接的防范措施。如果备份系统能够重新生成计算机的完整状态，那么我们将没有办法防止PRNG状态发生重复。理想地说，我们可以把备份系统配置成PRNG可知的，但可能会太多以至于问不过来。只要攻击者不把时钟重设为同一时间，种子文件与当前的时间在一起取Hash值就能够解决这个问题。如果备份系统中使用计数器来统计已经做了多少次恢复操作，那么同样的解决办法也适用。我们用种子文件与计数器一起取Hash值。这种方法还应该进一步研究，但因为它要根据具体平台而定，这里就不能给出整体的解决方案了。

10.6.5 文件系统的自动更新

与种子文件相关的另一个重要问题是文件系统的自动更新。对大多数计算机来说，如果写一个种子文件，那么仅有几个内存缓冲区得到更新。数据实际上是以后才写到磁盘上的。即便你指示操作系统把数据写入磁盘（假定这是可能的），也会有很多磁盘驱动器仅仅把数据放入缓冲区，而不能保证要等多长时间才将数据写入介质中。

只要从种子文件重新生成种子，都必须要在用户请求随机数据之前更新文件。换句话说，我们必须要绝对保证数据已经在介质上被修正过了。当考虑到很多文件系统分别管理文件数据和文件管理信息时，情况变得更复杂了，因此重写种子文件也许会使文件管理数据暂时不一致。如果在这期间敌手攻击的话，我们可能得到一个被攻破的种子文件或完全失去种子文件。现在还没有一个好的办法来构建安全的系统。

有一些文件系统用日志来解决这些问题中的一部分。这种技术最初起源于大型数据库系统。日志是文件系统中所有已更新文件的顺序列表。当正常使用时，日志能保证更新总是一致的。从可靠性角度来说，这样的文件系统是更可取的。但遗憾的是，在一般的应用中，任何文件系统都不能完全保证更新何时被写入长久的存储器中。

只要硬件和操作系统不支持完全自动的和长久的文件更新，我们就不可能提出一个完美的种子文件解决方案。你需要详细考察你所工作的特定平台，尽你所能地以可靠的方法更新种子文件。

10.6.6 第一次启动

当我们第一次使用PRNG时，还不存在用于重新生成种子的种子文件。例如，使用一个新的PC机，它的操作系统是在工厂安装的。现在这个操作系统要产生一些安装所需的管理密钥，因此它需要使用PRNG。为了生产的方便，所有的机器都拥有一致的数据。没有初始的种子文件，因此我们不能使用它。我们可以等待足够多的随机事件来触发一次或多次重新生成种子进程，但那要花费很长的时间，并且我们也不知道何时能收集到足够的熵来产生好的密钥。

一个好的办法是，安装过程在配置时产生一个PRNG所需的随机种子文件。例如，可以用一个单独机器上的PRNG给每一个机器产生一个新的种子文件。或者是安装软件可以让测试者移动鼠标来收集一些初始的熵。选择何种方法要看具体的环境，但无论如何初始的熵是一定要提供的。不提供初始的熵就没有选择了。熵累加器要花很长时间给PRNG生成种子，而一些很重要的密钥在机器安装完成后不久就能通过PRNG产生了。

一定要记住，Fortuna累加器一旦有足够多真随机的熵，它就会给生成器产生种子。根据源实际上提供的熵的多少（这是Fortuna所不知道的）收集足够的熵来给发生器重新产生种子是要花很长时间的。用外部的随机源来产生第一个种子文件也许是最好的办法。

10.7 应该做什么

首先，不要相信你的程序语言或操作系统提供的任何随机数发生器，其中的大多数是不满足密码强PRNG的要求的。我们在本章所阐述的PRNG是一个重要的改进，因此我们更推崇Fortuna PRNG。正如前面所说，原因是显然的。

熵的收集总是很困难的，应该尽可能多得到一些源。Fortuna的主要优势就是低质量的源也不能损害系统，因此几乎没有理由不把它们加入。然而，得到熵源也许要修正操作系统驱动程序，这并不是很容易做到的。

种子文件是一个很容易理解的概念，但正确应用却有些困难。我们这里给了一些实际的建议，但大多数还要靠周围的环境来决定。你也许为了做好它，不得不改变操作系统甚至是硬件的一部分。为了得到安全性有时会影响到系统的其他部分，这是又一实例。尽自己的最大努力去做吧。祝你好运！

10.8 选取随机元素

PRNG产生随机字节序列。有时，这正是你需要的。另一种情况是，要从集合中选取一个随机元素。这就要注意一些问题。

无论何时选取随机元素，都要明确地假定元素是从均匀分布的特定集合中被随机地选取的（除非我们说明是另一种分布）。这就意味着每个元素被选取的概率是完全相同的①。它比你所想的要困难一些，很多程序都会出错。

令 n 是我们要选取的集合中元素的个数。我们这里仅讨论怎样从集合 $0, 1, \cdots, n-1$ 中选取一个随机元素。因为一旦能做到这一步，就能从大小是 n 的任何集合中选取元素。

如果 $n = 0$，说明没有元素被选取，因此这是一个简单的错误。如果 $n = 1$，你就没有其他选择，这又是一种简单的情形。如果 $n = 2^k$，那么说明你从PRNG中恰好得到 k 比特的随机数

① 如果我们设计的是128比特的安全级别，所能忍受的概率偏离是 2^{-128}。但这是很容易达到的。

据，你要把它们表示成在0, ⋯, n–1范围内的数字。这个数是随机均匀分布的。(你也许不得不从PRNG中得到整个的字节，然后丢弃最后字节的几个比特，保留k比特，但这是容易做到的。)

如果n不是2的幂怎么办呢?有一些程序是随机地选取32比特的整数，然后以n为模。但这种算法会引起最终结果的概率偏离。例如当$n = 5$时，定义$m := \lfloor 2^{32}/5 \rfloor$。如果我们随机选取一个均匀分布的32比特整数并取模5，那么结果是1，2，3和4的概率为$m/2^{32}$，结果是0的概率为$(m+1)/2^{32}$。概率偏离虽然很小，但也是很主要的。攻击者一定会很容易地在允许的2^{128}步之内检测到这种偏离。

正确的办法应该是，用尝试然后排错的方式在任意范围内选取随机数。为了产生范围在0, ⋯, 4内的随机数，我们首先要产生一个在0, ⋯, 7内的随机值，因为8是2的幂。如果结果是5或更大，我们就把它丢弃然后再选取0, ⋯, 7内的新的随机值。我们一直做下去，直到结果是在需要的范围内为止。换句话说，我们产生一个合适比特的随机数，把不正确的丢弃。

下面对怎样在0, ⋯, n–1($n \geqslant 2$)内选取随机数给出一个形式化的说明。

1. 令k是满足$2^k \geqslant n$的最小整数。
2. 用PRNG产生一个k比特的随机数K。这个数在范围0, ⋯, 2^k–1内。你也许为了达到此目的，要生成整个字节然后丢弃最后字节的一部分，但这是很容易的。
3. 如果$K \geqslant n$，回到第二步。
4. K是最后的结果。

这个过程有一些浪费，最坏的情形是丢弃一半的尝试。下面是一种改进。由于$2^{32}-1$是5的倍数，我们能在0, ⋯, $2^{32}-2$范围内选择一个随机数，然后取模5作为最后的结果。为了在0, ⋯, $2^{32}-2$内选值，我们用一个“低效”的尝试和丢弃的算法，但这时丢弃中间结果的概率是很低的。

整体的形式化表示是选择一个k满足$2^k \geqslant n$。定义$q := \lfloor 2^k/n \rfloor$。首先用尝试和丢弃的算法在0, ⋯, nq–1范围内选取一个随机数r。一旦合适的r产生了，最终的结果就是($r \bmod n$)。

现在我们还没有任何办法产生一个不是2的幂的均匀分布的随机数而不必丢弃一些比特。但这不是问题，给定一个合适的PRNG，就不会存在随机比特的缺陷了。

第11章 素　　数

接下来的两章阐释公钥密码系统。遗憾的是，这需要一点儿数学知识。人们常常不去理解这些数学知识，而只是看看数学公式和方程，我们认为这样做是很危险的一件事情。要使用一个工具，就必须了解这个工具的性质。对于像Hash函数之类的事情是容易做到的。我们有Hash函数的“理想”模型，而且要求实际的Hash函数也具有该理想模型下的性质。对于公钥密码系统就不那么容易了，因为没有相应的“理想”模型。在实际情况中，一定会涉及到公钥系统的数学性质，并且为了安全地实现就必须理解这些性质。这里没有捷径，你必须理解这些数学概念。这些数学概念没有那么困难，所需要的基础知识只是高等数学。

这一章讲述素数。素数在数学中扮演了重要的角色，而我们之所以对素数感兴趣，是因为大多数的公钥密码系统都是基于素数的。

11.1　可除性与素数

如果a除b没有余数，称数a是数b的一个因子（记为$a|b$，读做“a除b”）。例如，7是35的因子，故写成$7|35$。称一个数是素数，因为它恰好有两个因子，即1和它本身。例如13是一个素数，它的两个因子是1和13。最初的一些素数很容易找出来：2, 3, 5, 7, 11, 13, …。任意一个大于1而又不是素数的整数都称为合数。数1既不是素数又非合数。

我们要使用前几章中的专有数学符号和术语。这些符号乍看起来可能觉得困难和复杂，但是这一部分的数学概念却非常容易。

下面是有关可除性的一个简单引理：

引理1：如果，$a|b$，$b|c$那么$a|c$。

证明：如果$a|b$，那么存在整数s使得$as = b$（记住，b能够被a整除，所以它一定是a的倍数）。并且，如果$b|c$，那么存在整数t使得$bt = c$。然而这意味着$c = bt = (as)t = a(st)$，从而a是c的一个因子（要理解这一点只需验证每一个等式是正确的即可。得到的结论是，第一项c一定等于最后一项$a(st)$）。 □

这个引理陈述了一个事实。其证明论证了引理为何是正确的。小正方形符号（□）表示证明结束。数学家总喜欢使用大量的记号[①]。这是一个很简单的引理，只要记得符号$a|b$的意义，其证明就很容易理解。

① 使用符号有利也有弊。本书将采用我们认为最合适的方法进行论述。

数学家们对素数的研究已经历经了几个世纪。即使在今天，如果要生成所有小于100万的素数，也要使用2000年前的一个算法，它是由阿基米德的朋友Eratosthenes提出的（Eratosthenes也是第一个精确测量地球直径的人。至少在1700年后，当哥伦布计划由西航行至印度时，依其陈述使用了一个更小的——然而是错误的——地球的直径长度）。希腊的另一位数学家欧几里得给出了一个完美的证明，从而表明了存在无穷多个素数。这个证明非常漂亮，我们将把它写出来。阅读该证明将帮助你重新温习一下相应的数学知识。

在开始真正的证明之前，我们先给出一个简单的引理。

引理2： 设n是大于1的正数，设d是n的最小的因子并且d大于1，那么d是素数。

证明： 首先我们必须检查d是良好定义的（如果存在一个没有最小因子的数n，那么d就没有正确地定义，引理也就没有意义）。我们知道，n是n的因子，并且$n>1$，所以n至少有一个大于1的因子。因此，它一定存在大于1的最小因子。

为了证明d是素数，我们使用一个标准的数学技巧，称为归谬法或者反证法。要证明一个命题X，我们首先假定X是不正确的，然后说明这个假设导致一个矛盾。如果假定X不正确导致了矛盾，那么X显然一定是正确的。

在这里，我们假定d不是一个素数。如果d不是素数，它就有因子e满足$1<e<d$。然而根据引理1我们知道，如果$e|d$并且$d|n$，那么$e|n$，于是e是n的因子并且小于d。但这是一个矛盾，因为d定义为n的最小的因子。由于这个矛盾不可能是正确的，我们的假定一定是错误的，从而d必定是素数。 □

如果觉得这种类型的证明有一点儿迷惑，请不要担心，慢慢就会习惯了。

下面我们证明定理3。

定理3（欧几里得）： 存在无穷多个素数。

证明： 我们再一次假定要证明的命题的相反命题成立。这里假设素数的个数是有限的，因而素数的列表也是有限的。把它们记为$p_1, p_2, p_3, \cdots, p_k$，其中$k$是素数的个数。定义数$n := p_1p_2p_3\cdots p_k+1$，它是所有素数的乘积加1。

考虑n的大于1的最小因子，我们又把它称为d。那么d是素数（根据引理2）并且$d|n$。但是，在有限的素数列表中没有一个素数是n的因子。记住，它们都是$n-1$的因子，因此，如果用列表中的一个素数p_i去除n，所得余数总是1，所以d是素数并且不在该素数列表中。然而，这是一个矛盾，因为该列表定义为包含所有的素数。因此，假定素数的个数有限导致矛盾，留给我们的结论是“素数的个数是无限的”。 □

这基本上就是2000多年前欧几里得给出的证明。

关于素数的分布有更多的结果，但有意思的是给出特定区间内素数个数的公式却并不容易。素数的出现看起来相当随机，甚至有一些看起来简单的猜想从未得到证明。例如戈德巴赫

猜想：每一个大于2的偶数都是两个素数之和。利用计算机很容易验证这个猜想对于相对较小的偶数是成立的，但是数学家们还不知道对于所有的偶数是否成立。

了解算术基本定理也是很有用的：任意一个大于1的整数恰好有一种方式写成素数的乘积（不计所写素数的顺序）。例如，$15 = 3 \cdot 5$；$255 = 3 \cdot 5 \cdot 17$；$60 = 2 \cdot 2 \cdot 3 \cdot 5$。这里我们不打算证明这个定理。如果想了解详细的证明细节，可以查看任何一本数论教科书。

11.2 生成小素数

有时得到一个小素数列表是很有用的，所以我们在这里给出Eratosthenes筛选法，该筛选法仍然是最好的产生小素数的算法。

function SmallPrimeList
input: n　　待产生的素数的限制。必须满足 $2 \leqslant n \leqslant 2^{20}$。
output: P　　所有小于等于 n 的素数。
　限制 n 的大小。如果 n 过大，将导致内存溢出。
　assert $2 \leqslant n \leqslant 2^{20}$
　初始化一列都设置为1的标记。
　$(b_2, b_3, \cdots, b_n) \leftarrow (1, 1, \cdots, 1)$
　$i \leftarrow 2$
　while $i^2 \leqslant n$ **do**
　　我们已经找到一个素数 i。标记所有 i 的倍数为合数。
　　for $j \in 2i, 3i, 4i, \cdots, \lfloor n/i \rfloor i$ **do**
　　　$b_j \leftarrow 0$
　　od
　　寻找列表中的下一个素数。可以证明这一循环不会导致 $i > n$，该条件会使用一个不存在的 b_i。
　　repeat
　　　$i \leftarrow i + 1$
　　until $b_i = 1$
　od
　现在所有的素数都已经标记了一个1。把它们都收集在列表中。
　$P \leftarrow [\,]$
　for $k \in 2, 3, 4, \cdots, n$ **do**
　　if $b_k = 1$ **then**
　　　$P \leftarrow P \| k$

```
    fi
  od
  return P
```

这个算法基于一个简单的思想。任何合数c都能够被一个小于c的素数整除。对从1到n的每一个数，我们做一个标记列表。每个标记表示对应的数是否为素数。最初，所有的数都被标识为可能的素数，而把它的标记设置为1。我们从设置i为第一个素数2开始。当然，任何i的倍数都不是素数，于是我们标识$2i$、$3i$、$4i$等为合数，把它们的标记设置为0。然后我们增加i值直至找到另一个候选素数。现在这个候选者不会被任何更小的素数整除，否则它已经被标识为合数了，因此这个新的i一定是下一个素数。继续标识合数并找到接下来的素数，直至$i^2 > n$。

很明显，不会有素数被标识为合数，因为仅当我们知道一个数的因子时才把它标识为合数（标识为合数的环都是$2i, 3i, \cdots$的形式。其中的每一项都有一个因子i，因而不可能是素数）。

为什么当$i^2 > n$时就可以停下来了呢？假设一个数k是合数，令p表示它的大于1的最小的因子。我们已经知道p是素数（见引理2）。设$q := k/p$，则有$p \leqslant q$；否则q就是k的一个因子并且比p小，这与p的定义相矛盾。关键要注意到$p \leqslant \sqrt{k}$，因为，如果p比$\sqrt{k}$大，那么我们有$k = p \cdot q > \sqrt{k} \cdot q \geqslant \sqrt{k} \cdot p > \sqrt{k} \cdot \sqrt{k} = k$。最后一个不等式表示$k > k$，这显然是一个矛盾。所以$p \leqslant \sqrt{k}$。

我们已经说明了任何合数k都能被一个小于等于$\sqrt{k}$的素数整除。于是任何小于等于n的合数都能够被一个小于等于$\sqrt{n}$的素数整除。如果$i^2 > n$，那么$i > \sqrt{n}$。但是我们已经把小于i的所有素数的倍数标识为合数，因此任一个小于n的合数都已经按照这种方法被标识出来。在这个列表中仍然标识为素数的数就是真正的素数。

算法的最后一步只是把这些素数收集起来用于输出。

这个算法可以做一些优化，但我们把它们都省去了，为的是使本书更简单。如果正确地实现，这个算法是很快的。

你可能感到奇怪，为什么需要小素数。实际上，小素数对于生成大素数是很有用的，我们很快就会看到这一点。

11.3 素数的模运算

素数在密码学中之所以如此有用，主要原因是可以进行素数模运算。

设p是素数。当我们进行素数模运算时，只需要使用$0, 1, \cdots, p-1$这些数。素数模的基本规则是，把这些数作为整数进行计算，就如通常的计算一样，但是每一次的结果r都要对p进行模运算。取模运算很容易：只是用p除r，丢弃所得的商，而把余数作为答案即可。例如，如

果对 25 做模 7 运算，则要用 7 除 25，得到的商为 3，余数为 4。余数就是答案，于是(25 mod 7) = 4。记号(a mod b)用于确切地表示模运算，但是，由于模运算很常用，而数学家都相当“懒惰”，所以通常也使用其他的记号。整个方程的书写常常也没有体现任何模运算，只是把(mod p)加在方程的后面，以表示所有的计算都是取模p运算的。根据上下文，如果这种情况比较清楚，取模运算记号甚至可以省去，但是你自己必须知道这是模运算。

不需要在模运算两边加上圆括号。我们也可以写成 a mod b，但是模运算看起来很像一般的正文，对那些不太习惯于此的人可能觉得有一点儿迷惑。为了避免混淆，我们倾向于采用带圆括号的记法(a mod b)。

警告：任何整数取模 p 运算之后总在 $0, \cdots, p-1$ 范围之内，即使原来的整数是负的。一些编程语言有这样的性质：允许模运算的结果为负值（对此数学家感到非常恼火）。如果对 -1 做模 p 运算，其结果是 $p-1$。更一般的情况是：要计算(a mod p)，寻找整数 q 和 r 使 $a = qp + r$，并且 $0 \leqslant r < p$。(a mod p)的值就定义为 r。如果取 $a = -1$，那么 $q = -1$，而 $r = p-1$。

11.3.1　加法和减法

模 p 加法运算很简单，只需要把两个数相加，如果结果大于或者等于 p，就减去 p。由于两个输入都在 $0, \cdots, p-1$ 的范围内，二者的和不可能超过 $2p-1$，于是只需要做一次减去 p 的运算，得到的结果就在正确的范围之内了。

减法类似于加法。减去一个数，如果结果是负的就加上 p。

这些规则只有当两个输入都已经是模p的数时才有效。如果它们不在这个范围内，就必须做一次完整的模 p 约化。

需要一段时间才能习惯于模运算。假设有一个方程，如 $5 + 3 = 1 (\text{mod}) 7$。该方程乍看起来有点儿奇怪。你知道 5 加 3 不是 1。但是，尽管在整数运算中 $5 + 3 = 8$ 是正确的，做模 7 运算则有 $8 \bmod 7 = 1$，所以 $5 + 3 = 1 \pmod 7$。

模运算在实际生活中使用得非常频繁，只是我们没有意识到这一点。当计算一天的时间时，我们把小时做模 12 运算（或者模 24 运算）。公共汽车的时间表规定，汽车在每小时过 55 分钟离开，路上需要行驶 15 分钟。为了弄清楚公共汽车何时到达，我们计算 $55 + 15 = 10 \pmod{60}$，并确定它在每小时过 10 分钟到达。从现在开始，我们限定于模素数运算，然而你可以对任何你喜欢的数做模运算。

11.3.2　乘法

乘法总是比加法需要更多的工作量。为了计算(ab mod p)，首先把 a 和 b 当做整数计算 ab，然后对计算结果做模 p 运算。现在 ab 可以像 $(p-1)^2 = p^2-2p + 1$ 一样大。你必须执行一次长除法来找到(q, r)以满足 $ab = qp + r$，而且 $0 \leqslant r < p$。丢弃 q，r 就是答案。

让我们给出一个例子：设 $p = 5$。当计算 $3 \cdot 4 \pmod p$时结果是 2。毕竟 $3 \cdot 4 = 12$，而 $(12 \bmod 5) = 2$，于是得到 $3 \cdot 4 = 2 \pmod p$。

11.3.3 群和有限域

数学家称模素数 p 的数的集合为有限域，并经常称它为“mod p”域，或者只是“mod p”。关于“mod p”域的计算有一些有用的提示：

- 对于一个数 p，总是可以加上或者减去它的任何倍数而不会改变结果。
- 任何结果总是在 $0, 1, \cdots, p-1$ 范围之内。
- 可以在整数范围内做整个计算，只在最后时刻做模运算。于是所有的关于整数的代数规则（比如 $a(b+c)=ab+ac$）仍然适用。

在不同的书中，涉及对整数的有限域取模p时使用的记号是不同的。我们将使用记号Z_p表示对有限域取模 p。在其他书中你可能会看到 GF(p)甚至 Z/pZ。

我们还需要引进群的概念，这是另一个简单的数学术语。群只是一些数的集合，而且带有一种运算，比如加法或者乘法[①]。Z_p 中的数构成一个带加法的群。你可以把任意两个数相加，得到群中的另一个数。如果要在一个群中使用乘法，就不能使用0（这与以下事实有关：乘以0没有什么意义，而且不能够除以0）。然而，数 $1, \cdots, p-1$ 以及模 p 乘法构成一个群，这个群称为模p乘法群并且有多种方式的记法。我们将使用记号 Z_p^*。一个有限域包括两个群：加法群和乘法群。在 Z_p 情形中，这个有限域由模 p 加法定义的加法群和乘法群 Z_p^* 组成。

群可以包含子群。一个子群由整个群中的一些元素组成。如果对子群中的两个元素使用群运算，又会得到子群中的一个元素。这听起来有些复杂，所以举一个例子说明。模 8 的数以及加法(mod 8)构成一个群，数$\{0, 2, 4, 6\}$构成一个子群。可以把这些数中的任何两个数对 8 取模后相加，得到子群中的另一个元素。对乘法群也一样。模 7 的乘法群由数$\{1, \cdots, 6\}$以及模 7 乘法运算组成。集合$\{1, 6\}$构成一个子群，集合$\{1, 2, 4\}$也一样。你可以检查同一个子群中的任何两个元素模 7 相乘，是否得到该子群中的另一个元素。

我们使用子群来加快一些密码操作。它们也能够用于攻击一个系统，这就是需要知道它们的原因。

到目前为止，我们只是讨论了模素数的加法、减法和乘法。为了完整地定义乘法群，还需要乘法的逆运算：除法。事实证明，可以对模 p 的数定义除法。简单的定义是：$a/b \pmod p$等于满足 $c \cdot b = a \pmod p$的数 c。不能用 0 去除一个数，但是只要 $b \neq 0$，$a/b \pmod p$总是有意义的。

那么，如何计算两个数模p的商呢？这就复杂多了，需要几页篇幅的文字来解释。我们首先再次回忆一下 2000 多年前的欧几里得计算 GCD 的算法。

11.3.4 GCD 算法

大学数学的另一个新课程：两个数 a 和 b 的最大公因子（或GCD）是满足 $k|a$ 和 $k|b$ 的最大的整数 k。换句话说，gcd(a, b)是能够同时整除 a 和 b 的最大的数。

① 此外还有两个要求，但是我们使用的群都满足这些条件。

欧几里得给出了一个算法来计算两个数的GCD,这个算法在几千年后的今天仍然在使用。关于这个算法的详细讨论见 Knuth [55]。

function GCD
input: a 正整数。
b 正整数。
output: k a 和 b 的最大公因子。
assert $a \geqslant 0 \wedge b \geqslant 0$
while $a \neq 0$ **do**
$(a, b) \leftarrow (b \bmod a, a)$
od
return b

为什么这样就可以了呢？首先注意，这样的安排不会改变a和b的公因子的集合。实际上，($b \bmod a$)就是 $b-sa$，其中 s 为一个整数。任何同时整除 a 和 b 的整数 k 也能够同时整除 a 和 ($b \bmod a$)，反之亦然。而当 $a = 0$ 时，b 就是 a 和 b 的公因子，并且 b 显然是最大的公因子。可以自己检验一下这样的循环一定会结束，因为 a 和 b 都将变得越来越小直至到达 0。

作为例子，我们计算 21 和 30 的 GCD。我们由$(a, b) = (21, 20)$开始。第一轮计算$(30 \bmod 21) = 9$，于是得到$(a, b) = (9, 21)$。接下来我们计算$(21 \bmod 9) = 3$，于是有$(a, b) = (3, 9)$。最后一轮计算$(9 \bmod 3) = 0$，从而得到$(a, b) = (0, 3)$。该算法输出为 3，这确实是 21 和 30 的最大公因子。

对应于 GCD，我们有 LCM（或称最小公倍数）。a 和 b 的最小公倍数是同时为 a 和 b 的倍数的最小的数。比如，$\text{lcm}(6, 8) = 24$。GCD 和 LCM 是密切相关的，且满足如下方程：

$$\text{lcm}(a, b) = \frac{ab}{\gcd(a, b)}$$

对此我们不做证明，只是作为一个事实加以表述。

11.3.5 扩展欧几里得算法

上面所述并不能帮助我们计算模p除法。我们所需要的是扩展欧几里得算法。其思想是在计算 $\gcd(a, b)$时也找到满足 $\gcd(a, b) = ua + vb$的两个整数 u 和 v。这将使得我们能够计算 $a/b \pmod p$。

function extendedGCD
input: a 正整数变量。
b 正整数变量。
output: k a 和 b 的最大公因子。
(u, v)满足 $ua + vb = k$ 的整数。
assert $a \geqslant 0 \wedge b \geqslant 0$
$(c, d) \leftarrow (a, b)$

$(u_c, v_c, u_d, v_d) \leftarrow (1, 0, 0, 1)$

while $c \neq 0$ **do**

  不变量：$u_c a + v_c b = c \wedge u_d a + v_d b = d$

  $q \leftarrow \lfloor d/c \rfloor$

  $(c, d) \leftarrow (d - qc, c)$

  $(u_c, v_c, u_d, v_d) \leftarrow (u_d - qu_c, v_d - qv_c, u_c, v_c)$

od

return $d, (u_d, v_d)$

这个算法看起来很像GCD算法。我们引进了新的变量c和d来代替a和b，这是因为我们在不变量中要涉及到原始的a和b。如果只看c和d，这正是GCD算法（稍有不同的是我们重写了$d \bmod c$公式，但所得结果是一样的）。我们增加了4个保持给定不变量的变量，对每一个产生的c值或者d值，明确了如何表达该值为a和b的线性组合。在初始化中这是容易做到的，因为c初始化为a，d初始化为b。当修改c和d时，更新变量u和v并不十分困难。

为什么要使用扩展欧几里得算法呢？假设要计算$1/b \bmod p$，其中$1 \leqslant b < p$。我们使用扩展欧几里得算法来计算extendedGCD(b, p)。现在我们知道b和p的GCD是1，因为p是素数，从而没有其他的因子。但是extendedGCD函数也提供了两个数u和v以满足$ub + vp = \gcd(b, p) = 1$。换言之，$ub = 1 - vp$，或者$ub = 1 \pmod p$。这等于说$u = 1/b \pmod p$，即b模p的逆。除法a/b可以通过a乘u计算，于是得到$a/b = au \pmod p$，而我们知道如何计算最后一个公式。

扩展欧几里得算法允许我们计算模素数的逆，从而允许计算模p除法。同模p的加法、减法以及乘法一起，这允许我们计算有限域模p的所有4个基本运算。

我们注意到u可能是负的，所以在把它用做b的逆元之前进行模p约化或许是一个好主意。

如果仔细察看extendedGCD算法，会发现只需要输出u，可以不管变量v_c和v_d，因为它们不影响u的计算。这稍微减少了一些计算模p除法的工作量。

11.3.6 模2运算

一个有趣的特殊情形是模2计算。记住，2是素数，因此能够进行模运算。如果做过一些编程，可能会觉得熟悉。模2加法和乘法表如图11.1所示。模2加法就是编程语言中的异或函数（XOR），乘法正是简单的AND运算。在模2域中，只有一个可能的逆元（1/1 = 1），所以除法运算与乘法运算是相同的。你不应该感到奇怪，域Z_2是分析某些计算机算法的一个重要工具。

+	0	1
0	0	1
1	1	0

·	0	1
0	0	0
1	0	1

图11.1 模2加法和乘法

11.4　大素数

一些密码模块使用非常大的素数，我们说的是几百位的素数。不用担心，无需用手来计算这些素数，这是计算机做的事情。

要对任何这么大的数进行计算，需要一个多精度库。不能使用浮点数，因为它们不可能有几百位的精度。不能使用正常的整数，因为大多数编程语言都限于十几位数字。很少有编程语言支持任意精度的整数。编写执行大整数计算的程序非常吸引人。作为很好的综述，参见Knuth [55，4.3 节]。然而，实现一个多精度库需要的工作量比想像的要多，因为不仅需要得到正确的答案，而且还要尽快地计算出来。因为有相当多的特殊情形需要仔细处理，因此要节省时间以处理更重要的事情。你可以从因特网上很多的免费库中下载一个，或者使用像 Python 之类的具有内置的大整数支持的语言。

对于公钥密码学来说，我们需要产生的素数有 2000 ~ 4000 比特长。产生如此之大的素数的基本方法不可思议地简单：选择一个随机数并检查它是否为素数。有非常好的算法来确定一个大数是否为素数。在一个数 n 的附近，大约有 $1/\ln n$ 的数是素数。（n 的自然对数，或者简记为 $\ln n$，是科学计算中的一个标准函数。这个例子给出一个概念，即对于大的输入对数的增长有多慢：2^k 的自然对数比 $0.7 \cdot k$ 稍微小一点。）一个 2000 比特的数落在 2^{1999} 和 2^{2000} 之间。在这个范围内，每 1386 个数中就有一个素数，而且这包括大量的显然是合数的数，比如偶数。

产生一个大素数的过程如下：

function generateLargePrime
input: l　素数所在范围的下界。
　　　　u　素数所在范围的上界。
output: p　一个随机的素数，位于区间 $l, \cdots, u$。
　敏感范围检查。
　assert $2 < l \leqslant u$
　计算最大尝试次数。
　$r \leftarrow 100(\lfloor \log_2 u \rfloor + 1)$
　repeat
　　$r \leftarrow r-1$
　　assert $r > 0$
　　在正确的区间内随机选择 n。
　　$n \in_R l, \cdots, u$
　　继续尝试直至找到一个素数。
　until isPrime(n)
　return n

我们使用 $\in R$ 算子来表示从一个集合中随机地选取。当然，这需要有的输出来自 PRNG。

这个算法相对比较直观。首先检查我们得到的区间是否合理。$l \leq 2$ 和 $l \geq u$ 的情况是没有用的，而且会产生问题。注意边界条件：$l = 2$ 的情况是不允许的[①]。接下来计算我们需要做多少次尝试才能找到一个素数。有一些区间是不包含素数的。比如，在区间 90, …, 96 中就没有素数。无论输入是什么，一个合适的程序绝不该为此而中止，所以我们限制尝试的次数，如果超过这个数就输出错误。该尝试为多少次呢？如前面所说，在 u 的附近，大约每 $0.7 \log_2 u$ 个数中就有一个数是素数（函数 $\log_2$ 是以 2 为底的对数函数。最简单的定义是 $\log_2(x) := \ln x / \ln 2$）。$\log_2 u$ 这个数是难以计算的，但计算 $\lfloor \log_2 u \rfloor + 1$ 就容易多了，它是表示 u 为二元数所必需的比特数。所以，如果 u 是 2017 比特长的整数，那么 $\lfloor \log_2 u \rfloor + 1 = 2017$。因数 100 保证了不太可能找不到素数。对于足够大的区间，因运气差而导致失败的概率不超过 2^{-128}，所以我们可以忽略这种风险。同时，这个限制确实保证了函数 generateLargePrime 将会停止。在产生错误的断言上我们有一点儿草率，一个正确的实现在产生错误时会说明产生了什么错误。

主循环比较简单。在检查完尝试次数的限制之后，选择一个随机数并使用 isPrime 函数检查它是否为素数。很快我们就要定义这个函数。

要确保你所选择的数 n 在 $l, \cdots, u$ 范围内是均匀分布的。如果你想对素数保密，还要保证这个范围不能太小。如果敌手知道你所使用的区间，并且该区间内的素数少于 2^{128} 个，那么敌手就可以尝试所有的可能。

如果愿意的话，可以在产生一个候选者 n 之后，通过设置其最低位比特来保证所产生的随机数是奇数。由于 2 不在区间内，所以不会影响所选素数的概率分布，而且这会使你要做的努力减半。然而，这只有在 u 是奇数时是安全的，否则设置最低位比特可能正好使 n 跳到允许的范围之外。

函数 isPrime 是一个两步过滤器。第一步是简单的测试，设法用所有的小素数来除 n。这将很快地清除掉绝大多数合数，这些合数能够被小素数整除。如果找不到因子，应进行一个重要的测试，称为 Rabin-Miller 测试。

function isPrime
input: n 大于或等于 3 的整数。
output: b 是否为素数的布尔判断。
  assert $n \geq 3$
  for $p \in \{$所有素数$\leq 1000\}$ **do**
    if p 是 n 的因子，**then**
      return $p = n$

① 如果变量取 2，下面我们使用的 Rabin-Miller 算法就不会很好地工作。既然我们已经知道了 2 是素数，所以这里就不需要产生它了。

```
    fi
  od
  return Rabin-Miller(n)
```

如果不想生成小素数，可以这样做：不是尝试所有的素数，可以按照顺序尝试2和所有的奇数3, 5, 7, ···, 999。这个序列包括了1000以下所有的素数，但也包括了许多无用的合数。这个顺序很重要，能够保证一些小合数（比如9）正确地被检测为合数。1000这个界限是任意的，可以按照最优的性能进行选取。

接下来解释可以完成这项艰难工作的神奇的Rabin-Miller测试。

11.4.1　素性测试

事实上，检测一个数是否为素数是非常容易的。至少，同分解一个数找到其素数因子比起来，它是非常容易的。这些测试是不完善的，它们的完善性都是概率概念上的，因此存在一定的给出错误答案的概率。通过重复运行同一个测试，我们可以把错误概率降低到一个可接受的水平。

这里选择的素性测试是Rabin-Miller测试，这个测试的数学基础已经超出了本书的范围。这个测试的目的是确定一个奇数 n 是否为素数。选择一个不超过 n 的随机值 a，称为基，并检查 a 模 n 的某个性质，当 n 是素数时该性质总是成立的。然而，当 n 不是素数时，可以证明这个性质至多对25%的所有可能基值成立。通过对不同的随机值 a 重复这个测试，可以建立起对最终结果的信心。如果 n 是素数，它将始终被测试为素数；如果 n 不是素数，那么 a 值通过测试的可能最多有75%，而且通过多重测试的概率可以达到像你想要的那样小。我们把错误结果的概率限定为 2^{-128} 以得到所需要的安全等级。

算法如下：

```
function Rabin-Miller
input:  n    大于或等于 3 的奇数。
output: b    布尔判断，表示 n 是否为素数。
  assert n ≥ 3 ∧ n mod 2 = 1
  首先计算（s, t），s 是奇数并且 2^t s = n−1。
  (s, t) ← (n−1, 0)
  while s mod 2 = 0 do
    (s, t) ← (s/2, t + 1)
  od
  我们可以通过k获知错误结果的概率。这个概率至多为2^(-k)。进行循环直至错误结果的概
  率足够小为止。
  k ← 0
```

```
while k < 128 do
  随机选择满足 2 ≤ a ≤ n−1。
  a ∈ R 2, ⋯, n−1
  耗时的操作：模指数。
  v ← a^s mod n
  当 v = 1 时，数 n 就通过了基 a 的测试。
  if v ≠ 1 then
  序列 v, v^2, ⋯, v^(2^t) 一定结束于值 1，并且如果 n 是素数，最后一个不等于 1 的值一定是 n−1。
  i ← 0
  while v ≠ n−1 do
    if i = t−1 then
      return false
    else
      (v, i) ← (v^2 mod n, i + 1)
    fi
  od
 fi
 到达这一点时，n 已经通过了基 a 的测试。因而我们把错误结果的概率降低了 2^2，所以
 可以把 k 加上 2。
 k ← k + 2
od
return true
```

这个算法只对大于或等于3的奇数n有效，于是我们首先检查这一点。函数应该只对合适的变量调用这个函数，但是每一个函数都负责检查它自己的输入和输出。你永远不会知道将来的软件会有什么变化。

这个测试背后的基本思想就是Fermat小定理[①]。对于任何素数n和所有$1 \leqslant a < n$的a，有关系式$a^{n-1} \bmod n = 1$成立。要完全理解这个式子成立的理由，需要更多的数学知识，这里就不解释了。有一个简单的测试（又称为Fermat素性测试），对许多随机选择的a值验证这一关系。遗憾的是，存在一些称为Carmichael的令人厌烦的数。这些数是合数，但（几乎）对所有的基a都通过了Fermat测试。

Rabin-Miller测试是Fermat测试的一种变型。首先我们记$n-1$为$2^t s$，其中s是奇数。如果要计算a^{n-1}，可以先计算a^s，然后把结果做t次平方就得到了$a^{s \cdot 2^t} = a^{n-1}$。此刻，如果$a^s = 1 \pmod n$，

① 有几个定理都以Fermat的名字命名。Fermat大定理是最有名的一个，它涉及到方程$a^n + b^n = c^n$和在这里写不下的证明。

那么重复平方不会改变结果，所以有 $a^{n-1}=1 \pmod n$。如果 $a^s \neq 1 \pmod n$，那么我们观察数 $a^s, a^{s\cdot 2}, a^{s\cdot 2^2}, a^{s\cdot 2^3}, \cdots, a^{s\cdot 2^t}$（当然，都要模 n）。如果 n 是素数，我们知道最后一个数一定是 1。若 n 是素数，那么满足 $x^2=1 \pmod n$的数只有 1 和 $n-1$①。所以只要 n 是素数，那么该序列中必有一个数是$n-1$，否则我们绝不能让最后一个数等于 1。这正是 Rabin-Miller 测试要检验的。假如任何 a 的选取说明了 n 是合数，应立即返回。如果 n 继续被测试为素数，我们就对不同的 a 值继续测试，直至产生错误回答。而声称一个合数为素数的概率小于 2^{-128}。

如果对一个随机数使用这个测试，这个测试失败的概率要比我们采用的界限小很多。对几乎所有的合数 n，几乎所有的基值都将显示 n 是合数。你会发现有很多依赖于此的程序库只对大约5个或10个基执行测试。这个想法很好，尽管我们必须对需要多少次尝试才能达到2^{-128}左右的错误等级进行分析，但是这只对随机选取的数应用 isPrime 测试才成立。后面我们将遇到一些情形，其中要对从别人那里接收到的数使用素性测试。这些数可能是恶意选取的，所以 isPrime 函数本身必须达到 2^{-128} 的错误界。

当要对从别人那里接收到的数做测试时，进行完整的64次Rabin-Miller测试是必需的。当我们设法随机产生素数时，这样做就矫枉过正了。但产生素数时，大部分时间都花在拒绝合数上了（几乎所有的合数都被你做的第一次Rabin-Miller测试拒绝了）。由于在找到素数之前可能需要尝试几百个数，对最终的素数做 64 次测试只是比 10 次测试慢了一点儿而已。

在本章以前的版本中，Rabin-Miller 程序有第二个变量，用于选择最大的错误概率。但那是无用选项的一个很好的例子，所以我们把它去掉了。在任何时候，做一个好的测试达到 2^{-128} 界是很简单的，不正确使用的可能性很小。

然而还是存在 2^{-128} 的可能性，此时 isPrime 函数将给出错误的答案。实际上这个可能性非常小，你被陨星砸着的可能性也比它大很多，所以不要担心。

11.4.2　评测能力

Rabin-Miller测试的大多数时间都花在计算 $a^s \bmod n$ 上。不能先计算 a^s 然后再对 n 取模。世界上甚至没有计算机具有足够的内存来存储 a^s，更不用说具有足够的计算能力来计算它了：a 和 s 可以是几千比特长，但是我们只需要 $a^s \bmod n$。我们可以对所有的中间结果使用 mod n，从而防止这些数变得太大。

有多种方法计算 $a^s \bmod n$，但这里有一个简单的描述。为了计算 $a^s \bmod n$，我们使用下面的规则：

- 如果 $s=0$，那么答案是 1。
- 如果 $s>0$ 并且 s 是偶数，那么使用这些规则先计算 $y := a^{s/2} \bmod n$。所要的答案就是 $a^s \bmod n = y^2 \bmod n$。

① 容易验证 $(n-1)^2 = 1 \pmod n$。

- 如果 $s>0$ 并且 s 是奇数，那么先使用这些规则计算 $y := a^{(s-1)/2} \bmod n$。所要的答案就是 $a^s \bmod n = a \cdot y^2 \bmod n$。

这就是所谓的二元算法的一个递归公式。如果看看这些执行的操作，它从指数的最高位部分到最低位部分，一点一点地建立起我们想要的指数。把这个递归算法转换成循环也是可能的。

计算 $a^s \bmod n$ 需要多少次乘法？设 k 是 s 的比特数，即 $2^{k-1} \leqslant s < 2^k$。那么这个算法至多需要 $2k$ 次模 n 乘法。这不是太糟糕。若要对一个 2000 比特的数检测素性，则 s 大约也是 2000 比特长，我们只需要 4000 次乘法。虽然这还是有大量的工作量，然而却在大多数台式计算机的计算能力之内。

很多公钥密码系统都使用了这类模指数。任何好的多精度库都有用于计算模指数的优化方法。一个特殊的乘法称为 Montgomery 乘法将很适合这个任务。也有一些计算 a^s 的方法使用更少的乘法[10, 第 4 章]。每一个这样的技巧可以节省 10% ~ 30% 的模指数计算时间，所以，该乘法被结合起来使用很重要。

直截了当的模指数实现常常容易遭受计时攻击，详细情况以及可能的补救措施参见16.3节。

第12章 Diffie-Hellman

我们将遵循历史的发展来介绍公钥密码学。公钥密码学真正开始于Whitfield Diffie和Martin Hellman于1976年发表的“密码学中的新方向”（“New Directions in Cryptogr-aphy”）一文[22]。

到目前为止，我们只谈到了使用共享的密钥进行加密和认证。但是，我们从哪里得到这些共享的密钥呢？如果要和10个朋友进行通信，你可以与他们都会面并与每一个朋友交换一个密钥以备将来使用。然而，同所有的密钥一样，这些密钥应该定期更新，于是你就必须再次与所有的朋友会面并交换所有的密钥。10个朋友的群组一共需要45个密钥。如果这个群组变得更大一些，这些密钥的数量的增长是二次的。对于100个需要互相通信的人们，需要4950个密钥。这会很快变得难以管理。

Diffie和Hellman提出了这样一个问题：是否能够以有效的方式做到这一点。假定有一个加密密钥和解密密钥不相同的加密方案，你可以公布加密密钥，而对解密密钥保密。现在任何人都可以给你发送一个加密的消息，而只有你自己能够解密它。这样就可以解决上述必须分发很多不同密钥的问题。

Diffie和Hellman提出了这个问题，但是他们只能给出一个不完全的答案。他们的解决方案就是如今熟知的Diffie-Hellman密钥交换协议，通常简记为DH协议[22]。

DH协议是一个很好的思想。这个协议使得在不安全线路上通信的两个人以这样的方式协商一个密钥：两个人都能得到相同的密钥，并且这个密钥不会泄露给监听二人会话的人。

12.1 群

如果已经读过了上一章，你就不会对所涉及的素数感到奇怪了。在本章的其余部分中，p是一个大素数。可以认为p有2000比特到4000比特长。本章中的大多数计算都将在模p的意义下进行——在很多地方我们将不再明确说明这一点。DH协议使用模p的乘法群$\mathbb{Z}_p^*$，我们在11.3.3节讨论过。

在这个群中选择任意一个元素g，考虑数$1, g, g^2, g^3, \cdots$，当然所有的数都要模p。这是一个无限序列，但是$\mathbb{Z}_p^*$中只有有限序列（我们注意到$\mathbb{Z}_p^*$是由具有模p乘法运算的数$1, \cdots, p-1$构成的集合）。从某一点起这些数必定开始重复。我们假定从$g^i = g^j$处开始重复，其中$i < j$。由于在模p的意义下可以做除法，我们在两边除以g^i得到$1 = g^{j-i}$。换句话说，存在一个数$q := j-i$使得$g^q = 1 (\bmod p)$。我们把满足$g^q = 1 (\bmod p)$的最小正整数q称为g的阶。

如果继续乘以 g，就得到数列 $1, g, g^2, \cdots, g^{q-1}$。之后，$g^q = 1$ 序列开始重复。我们把 g 称为生成元，它产生集合 $1, g, g^2, \cdots, g^{q-1}$。能够写成 g 的方幂的元素个数的恰好是 q，即 g 的阶。

模 p 乘法的一个性质是，至少有一个 g 生成整个群。也就是说，至少存在一个 g 值，其阶 $q = p-1$。因此，可以把 $\mathbb{Z}_p^*$ 看做数列 $1, g, g^2, \cdots, g^{p-2}$，而不是数 $1, \cdots, p-1$。生成整个群的 g 称为这个群的本原元。

其他的 g 值能够生成小一些的集合。我们注意到，如果把 g 生成的集合中的两个元素相乘，我们得到 g 的另一个方幂，也就是该集合中的另一个元素。复习一下相关的数学知识，即知 g 生成的集合是另一个群。即：可以在这个群中做乘法和除法，正如模 p 在大群中一样。这些小群称为子群（见 11.3.3 节），它们在各种攻击中是很重要的。

最后还有一件事情要解释。对任意的元素 g，其阶是 $p-1$ 的因子。这一点不难理解。选择 g 为本原元，设 h 为其他任意的元素。由于 g 生成整个群，存在一个 x 使得 $h = g^x$。现在考虑由 h 生成的元素，它们是 $1, h, h^2, h^3, \cdots$，这等于 $1, g^x, g^{2x}, g^{3x}, \cdots$（当然，所有的计算仍然要模 p）。h 的阶是满足 $h^q = 1$ 的最小的 q，这等于说它是满足 $g^{xq} = 1$ 的最小的 q。对于任何 t，$g^t = 1$ 相当于 $t = 0 \pmod{p-1}$。于是 q 是满足 $xq = 0 \pmod{p-1}$ 的最小的 q。这恰好在 $q = (p-1)/\gcd(x, p-1)$ 时发生，所以 q 显然是 $p-1$ 的因子。

这里有一个简单的例子。让我们选择 $p = 7$。如果取 $g = 3$，那么 g 就是一个生成元，因为 $1, g, g^2, \cdots, g^6 = 1, 3, 2, 6, 4, 5$（这里再次提醒读者，所有的计算都要模 p）。元素 $h = 2$ 生成一个子群 $1, h, h^2 = 1, 2, 4$，因为 $h^3 = 2^3 \bmod 7 = 1$。元素 $h = 6$ 生成子群 1，6。这些子群的阶分别为 3 和 2，它们都是 $p-1$ 的因子。

这也部分地解释了 11.4.1 节中所介绍的素性测试。素性测试基于对任意的 a 有 $a^{p-1} = 1$，这很容易检验。假设 g 是 $\mathbb{Z}_p^*$ 的生成元，并且 x 满足 $g^x = a$。当 g 是整个群的生成元时，总存在上述 x。但现在 $a^{p-1} = g^{x(p-1)} = (g^{p-1})^x = 1^x = 1$。

12.2 基本的 DH

对于基本的 DH 协议，我们首先选取一个大素数 p 及其本原元 g，g 生成了整个群 $\mathbb{Z}_p^*$。在这个协议中，p 和 g 都是公开的常数，我们假定包括攻击者在内的所有参与方都知道它们。协议如图 12.1 所示，这是我们书写密码协议的一个常用的方式。该协议涉及的两个参与方为 Alice 和 Bob，时间按照从上到下的顺序进行。首先 Alice 选择 $\mathbb{Z}_p^*$ 中的一个随机数 x，这与在 $1, \cdots, p-1$ 中选择随机数是一样的。她计算 $g^x \bmod p$ 并把结果发送给 Bob。然后 Bob 在 $\mathbb{Z}_p^*$ 中选择一个随机数 y。他计算 $g^y \bmod p$ 并把结果发送给 Alice。最终的结果 k 定义为 g^{xy}。Alice 可以通过计算她从 Bob 那里得到的 g^y 的 x 次幂而得到 k。（中学数学：$(g^y)^x = g^{xy}$。）同样，Bob 计算 k 为 $(g^x)^y$。最后他们都得到了相同的 k，它可以用做密钥。

那么攻击者是怎样的呢？攻击者能够看到 g^x 和 g^y，但是不知道 x 或者 y。给定 g^x 和 g^y 来计算 g^{xy} 的问题就称为 Diffie-Hellman 问题，或者简称为 DH 问题。只要正确地选取 p 和 g，没有

有效的算法可以计算DH问题——至少我们不知道有这样的算法。已知的最好的方法是先由 g^x 计算出 x，之后攻击者就可以像 Alice 所做的那样计算 k 为 $(g^y)^x$。在实数中，由 g^x 计算 x 就称为对数函数，这个函数在任何科学计算器里都能看到。在有限域 $\mathbb{Z}_p^*$ 中，它称为离散对数；而在有限群中，由 g^x 计算 x 的问题通常称为离散对数问题，或者 DL 问题。

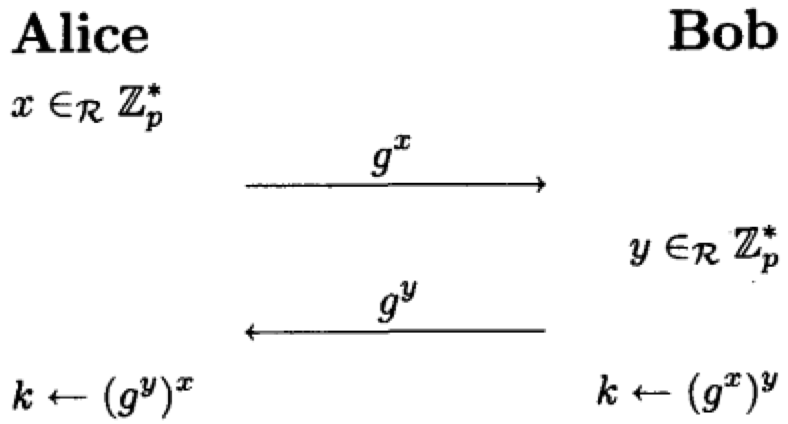


图 12.1　原始的 Diffie-Hellman 协议

原始的DH协议有多种使用方式，我们所写的是双方之间交换消息的使用方式。另一种使用方式是让每个人选择一个随机数 x，并在一个“数字”电话本中公布 $g^x \pmod p$。如果 Alice 要与Bob安全通信，她首先从电话本上得到 g^y，然后使用她自己的 x 计算 g^{xy}。同样，Bob 不用与 Alice 交互也可以计算 g^{xy}。这样的系统适用于没有直接交互的情况，如 E-mail 系统。

12.3　中间相遇攻击

DH协议不能防止的攻击是中间相遇攻击。让我们回头再看看这个协议。Alice知道她在与某一个人进行通信，但是她不知道是在与谁进行通信。Eve可以坐在协议的中间，当与Alice通信时假装成Bob，而与Bob通信时假装成Alice，这种情形如图 12.2 所示。对于 Alice 来说，这个协议看起来就像原始的 DH 协议。Alice 没有办法察觉出她是在与 Eve 而不是 Bob 通信。对于Bob来说也是一样的。只要 Eve 愿意，她可以一直这样伪装下去。假设 Alice 和 Bob 使用他们所认为的已经建立起来的密钥开始通信。Eve所需要做的就是转发Alice和Bob的所有通信。当然，Eve 必须解密从 Alice 那里得到的使用密钥 k 加密的所有数据，然后再使用密钥 k' 加密并发送给 Bob。对相反方向的通信她也需要做相同的事情，但这并没有多大的工作量。

若使用数字电话簿，这种攻击就困难多了。当每个人发送 g^x 时，只要电话簿的发行者验证他们的身份，Alice 就知道她是在使用 Bob 的 g^x。在本书的后面讨论数字签名和PKI时，我们再讨论其他的解决方案。

有一种情况不需要更多的基础设施就可以解决中间相遇攻击问题。如果密钥 k 用于加密电话交谈（或者视频连接），Alice 可以与 Bob交谈，通过声音就可以认出他。设 h 是某种 Hash 函数，如果Bob读取 $h(k)$ 的开始几位数并发给 Alice，那么 Alice 就可以验证 Bob 使用的密钥是否与自己使用的相同。Alice 读取 $h(k)$ 的接下来的几位数并发给 Bob，使 Bob 也能做同样的验证。

这种办法是有效的，但是只能用于密钥k的信息能够与通信另一方的真实身份绑定起来的情形。在大多计算机通信中，这个方法是不可行的。而且，如果Eve曾成功地建立一个语音合成器来模拟Bob，那么所有这些都将崩溃。最后，这个方法最大的问题在于，需要对用户进行训练。但是用户常常忽略这些安全程序。

Alice		Eve		Bob
$x \in_{\mathcal{R}} \mathbb{Z}_p^*$				
	$\xrightarrow{g^x}$			
		$v \in_{\mathcal{R}} \mathbb{Z}_p^*$		
			$\xrightarrow{g^v}$	
				$y \in_{\mathcal{R}} \mathbb{Z}_p^*$
			$\xleftarrow{g^y}$	
		$w \in_{\mathcal{R}} \mathbb{Z}_p^*$		
	$\xleftarrow{g^w}$			
$k \leftarrow (g^w)^x$		$k \leftarrow (g^x)^w$		
		$k' \leftarrow (g^y)^v$		$k' \leftarrow (g^v)^y$

图 12.2 有中间人的 Diffie-Hellman 协议

12.4 缺陷

实现DH协议有点儿棘手。比如，假如Eve拦截了通信并把g^x和g^y都替换为数1，那么Alice和Bob最终都将得到密钥$k=1$。对于这个结果，密钥协商协议看起来好像成功地完成了，但是Eve却知道了所产生的密钥。这是很糟糕的，我们必须采取某种方法来防止这种攻击。

第二个问题是，如果生成元g不是$\mathbb{Z}_p^*$的本原元，而只生成一个小的子群。也许g的阶为100万，此时集合$\{1, g, g^2, \cdots, g^{q-1}\}$只包含100万个元素。由于$k$在这个集合中，Eve可以容易地搜索到正确的密钥。很明显，一个必要条件就是g必须有很高的阶。但是，是谁选择的p和g呢？所有用户都在使用相同的值，所以大多数人都是从别人那里得到这些值的。为了安全起见，他们必须验证p和g的选取是正确的。Alice和Bob都应该检查以确保p是素数，g是模p的本原元。

模p的子群形成了另一个问题。Eve的替换g^x为数1的攻击是容易被挫败的，只要让Bob检查这一点就可以了。但是，Eve也可以把g^x替换为数h，其中h的阶比较小。现在Bob得到的密钥来自于由h生成的小集合，Eve可以尝试所有的可能来找到k（当然，Eve也可以对Alice实施同样的攻击）。Alice和Bob都必须做的就是验证该数不会生成小的子群。

让我们关注一下子群。模素数操作时，所有的（乘法）子群都可以由一个元素生成。整个群$\mathbb{Z}_p^*$由元素$1, \cdots, p-1$组成，一共有$p-1$个元素。每一个子群都具有$1, h, h^2, h^3, \cdots, h^{q-1}$的形

式，其中 h 是某一个元素，而 q 是 h 的阶。正如我们以前讨论过的，事实上 q 一定是 $p-1$ 的因子。换言之，任何子群的大小都是 $p-1$ 的因子。反之也成立：对任何 $p-1$ 的因子 d，都存在一个大小为 d 的子群。如果不想要小的子群，我们必须避免 $p-1$ 有小的因子。

这是一个问题。如果 p 是一个大素数，那么 $p-1$ 总是偶数，从而能被 2 整除。所以存在一个包含两个元素的子群，它由元素 1 和 $p-1$ 组成。但是除了这个总是存在的子群之外，我们能够避免其他的小子群，只要 $p-1$ 没有其他的小因子即可。

12.5 安全素数

我们的办法是使用安全素数 p。安全素数是一个（足够大的）形如 $2q+1$ 的素数 p，其中 q 也是素数。现在乘法群 $\mathbb{Z}_p^*$ 有下面的子群：

- 只有数 1 组成的平凡子群。
- 大小为 2 的子群，由 1 和 $p-1$ 组成。
- 大小为 q 的子群。
- 大小为 $2q$ 的整个群。

前两个子群容易避免。第三个是我们想要的子群。整个群存在另一个问题。考虑所有能够写成另一个数的平方（当然要模 p）的模 p 数的集合。事实上，$1,\cdots,p-1$ 中恰好有一半的数是平方数，而另一半是非平方数。整个群的任一个生成元都是非平方数（如果它是一个平方数，那么它的任何次幂都不可能产生一个非平方数，所以它不可能生成整个群）。

有一个数学函数称为Legendre符号，它能够确定一个模 p 数是否是平方数，而不需要找到这个数的根。存在一些有效的算法来计算Legendre符号。所以，如果 g 是一个非平方数而你把 g^x 发送出去了，那么任何观察者，包括 Eve，马上就能够确定 x 是奇数还是偶数。如果 x 是偶数，那么 g^x 是平方数；如果 x 是奇数，那么 g^x 是非平方数。由于Eve 可以使用Legendre 符号函数来确定一个数的平方性，所以她能够确定 x 是奇数还是偶数。这是一个异常的行为，Eve 除了最低位比特外不能知道 x 的值。避免这个问题的方法是只使用模 p 的平方数。这些数正好是阶为 q 的子群。另一个很好的性质是 q 是素数，于是我们不用担心它有更深一层的子群。

下面说明如何使用安全素数。选择 (p,q) 使得 $p=2q+1$，并且 p 和 q 都是素数（可以在试错法基础上利用 isPrime 函数来做这件事）。在 $2,\cdots,p-2$ 中选择一个随机数 α，并令 $g=\alpha^2 \pmod p$，检验 $g\neq 1$ 并且 $g\neq p-1$（如果 g 等于这些禁止的值，选择另一个 α 并再次尝试），这样产生的参数集合 (p,q,g) 就适合于 Diffie-Hellman 协议了。

Alice（或 Bob）每一次收到一个应该是 g 的方幂的值时，必须检查收到的这个值的确位于 g 生成的子群中。当使用上面描述的安全素数时，可以使用 Legendre 符号函数检查正确的子群成员资格。也有一个简单但是更慢一些的方法。当且仅当 $r^q=1 \pmod p$ 时数 r 是平方数。这时也要禁止使用数 1，因为它的使用总会导致一些问题。所以完整的检验就是：$r\neq 1 \wedge r^q \bmod p = 1$。

12.6 使用较小的子群

使用安全素数方法的缺点在于效率不高。如果素数p有n比特长，那么q就是n–1比特长，所以所有的指数都是n–1比特长。平均求幂运算将大约需要$3n/2$次模p数的乘法运算。对于大的素数p来说，这是相当大的工作量。

标准的解决办法是使用小一点的子群，下面是具体做法：选择一个256比特的素数q（换言之：$2^{255} < q < 2^{256}$），然后找到一个更大的素数p以满足$p = Nq + 1$，其中N是某个任意的值。要做到这一点，在适当的范围内随机选择一个N，计算p为$Nq + 1$，并检查p是否为素数。由于p一定是奇数，容易看出N必须是偶数。素数p可以有几千比特长。

然后需要找到一个阶为q的元素。我们使用类似于安全素数情形的方法。在$\mathbb{Z}_p^*$中选择一个随机数α，令$g := \alpha^N$。现在检查$g \neq 1$并且$g^q = 1$（由于q是奇数，第二个检验条件涵盖了$g = p-1$）。如果g不满足这些条件，就选择另一个α并再次尝试。这样产生的参数集合(p, q, g)将适用于Diffie-Hellman协议。

当我们选择这个小子群时，Alice和Bob交换的所有数值都在这个g生成的子群中。但是Eve能够进行干涉并将其替换为一个完全不同的值，因此，Alice或者Bob每一次收到一个应该位于g生成的子群中的值时，应该检查这个值是否的确位于该子群中。这个检查与安全素数情形下是一样的。如果$r \neq 1 \wedge r^q \bmod p = 1$，那么$r$就在正确的子群中。当然，他们也必须检查$r$并不在模$p$数的集合之外，所以完整的检验变成了$1 < r < p \wedge r^q = 1$。

由g生成的子群中的所有的数r都满足$r^q = 1$，所以，如果要计算r的e次幂，只需要计算$r^{e \bmod q}$，如果e比q大很多，那么计算的工作量要少很多。

那么，使用子群在效率上提高了多少呢？大素数p至少为2000比特长。在安全素数情形中，计算一般的g^x需要大约3000次乘法。在子群情形中，计算g^x需要大约384次乘法，因为x可以模q约化从而只有256比特长，这样节省将近7/8的工作量。当p变得更大时，会节省更多。这就是子群被广泛应用的原因。

12.7 p的长度

为DH系统的参数选择正确的长度很困难。到目前为止，我们的必要条件是：攻击者必须耗费2^{128}步来攻击系统。对于所有的对称密钥模块来说，这是一个容易实现的目标。启动像DH系统之类的公钥操作要昂贵得多，而且要得到想要的安全等级，计算代价的增长也非常迅速。

如果保持我们的安全需求，强制攻击者耗费2^{128}步来攻击系统，素数p应约为6800比特长。在目前的系统中，从性能的角度来看，这是一个现实问题。

对称模块的密钥长度和DH之类的公钥模块的密钥长度之间有很大的差异。从没有人把对称密钥长度（比如128比特或256比特）与公钥长度（可能是几千比特）进行过比较。公钥长度总是比对称密钥长度大很多[①]。

公钥操作比我们之前讨论的加密和认证函数要慢很多：在大多数系统中，对称密钥操作是无关紧要的，然而公钥操作对性能却有实质性的影响。因此，我们必须仔细地研究公钥操作的性能。

在一个系统中，对称密钥的长度通常是固定的。一旦设计的系统使用了特定的分组密码和Hash函数，也就固定了密钥的长度。这意味着在系统的生存期中对称密钥的长度都是固定的。另一方面，公钥长度几乎总是变化的，这使得变换密钥的长度相当容易。我们开始要设计能够使用30年的系统，而数据在首次处理之后需要保持20年的安全性。对称密钥的长度必须选取得足够大，使得它能够保护数据达50年之久。但是，可变长度的公钥只需要在未来的20年内保护数据。毕竟所有的密钥都有有限的使用寿命，公钥的有效期可能只有一年，但它应该能够在随后的20年内保护数据。这意味着公钥只需要保护数据21年，而不是对称密钥协议的50年。因为每年都产生一个新公钥，因此可以随着计算机技术的需要选择更大的公钥。

关于需要多大的素数p，最后的估计可以在文献[63]中找到。一个2048比特的素数应该保护数据到2022年前后，3072比特的素数到2038年之前是安全的，而4096比特的素数可以使用到2050年。上面提到的6800比特是由[63]中使用的同一个公式推导出来的。如果想要迫使攻击者执行2^{128}步才能完成一次攻击，p的长度就应该是6800比特。

要非常小心地看待这些预言。这些数的得来是有一些道理的，但是预言未来总是危险的。我们也许能对未来10年的密钥长度给出一些明智的预测，但是预言未来50年的事情就实在有点儿无聊了。不妨比较一下目前的计算机和密码学发展水平与50年前的技术水平。文献[63]中的预言是目前为止最好的估计，但是不要对它寄予过高的信任。

那么我们该怎么做呢？作为密码设计者，我们必须选择一个至少在未来20年是安全的密钥长度。显然，2048比特是一个下界。密码长度越大越好，但大密钥有明显的额外代价。面对如此多的不确定性，我们愿意保守一些。下面是我们的建议：使用2048比特作为绝对最小值（不要忘记，这个最小值会随着时间的流逝而增长）。如果完全从性能的角度来看，可以使用4096比特或者接近于4096比特的长度。此外，绝对要保证你的系统可以处理长达8192比特的密钥长度。如果对公钥系统的攻击出现了意外的突破，这样做将会扭转败局。密码分析的进步很大，可能会导致对小密钥长度的攻击。当系统在攻击范围之内时，这样做就可以转变成使用更大的密钥长度。它会耗费一些性能，但是系统的基本操作保留了下来。这比损失安全性和重新建造系统好多了。

有些应用要求数据的保密时间不止20年。在这种情况下，就需要使用大密钥。

① 这对于本书讨论的公钥方案都成立。其他的公钥方案，比如基于椭圆曲线的方案，可以有完全不同的密钥长度参数。

12.8 实践准则

下面是我们的一些实践准则，用于建立DH协议使用的子群。

选择q为一个256比特的素数（有一些针对DH指数的碰撞类型的攻击，于是所有的指数都应该是256比特长，以迫使攻击者至少要耗费2^{128}次操作）。选择p为一个形如$Nq+1$的大素数，其中N是整数（关于p的大小参见12.7节的讨论。计算相应的N的范围是平凡的）。随机选择一个g满足$g \neq 1$并且$g^q = 1$（一个容易的办法是选择一个随机的α，令$g = \alpha^N$，并检查g是否合适。如果g不满足这个标准，就尝试另一个α）。

接收到这个子群描述(p, q, g)的任何一个参与方都应该检验：

- p和q都是素数，q为256比特长，并且p足够大（不要信任太小的密钥）。
- q是$(p-1)$的因子。
- $g \neq 1$而且$g^q = 1$。

即使这些描述是由可信的来源提供的，也要进行这些检查。你可能会对系统常常是以某种奇怪的方式失败的（尤其是当它们遭受攻击时）感到吃惊。检查集合(p, q, g)要花一点儿时间，但在大多数系统中，同一个子群要用很长时间，所以这些检查只需要执行一次。

无论任何时候，当参与方收到一个应该位于子群中的数r时，都要验证$1 < r < p$和$r^q = 1$。注意，$r = 1$是不被允许的。

使用这些准则，我们得到如图12.3所示的Diffie-Hellman协议版本。双方开始时都要检查群参数。每一方只需要在协议启动时做一次检查，而不是每一次运行DH协议时都要检查。（然而，在每一次重新启动或者重新初始化之后都应该进行检查，因为这些参数可能会改变。）

协议的其他部分基本上与图12.1中的原始DH协议是一样的。现在Alice和Bob都使用该子群，于是两个指数x和y都在$1, \cdots, q-1$之内。Alice和Bob都要验证他们所接收的数是否在正确的子群中，以避免Eve发动的小子群攻击。

用于这个检查的记号是上面标记有问号的关系运算子。这表明Alice（或Bob）应该检查该关系是否成立。如果成立，那么一切顺利。如果该关系不正确，那么Alice当然认为她受到了攻击。标准的做法是：终止协议的执行，不发送任何其他的消息，并毁掉所有的相关协议数据。例如，在这个协议中，如果最后一步检查失败，Alice应该销毁x和Y。关于如何处理这些失败，请参见14.5.5节的详细讨论。

这个协议描述了一个安全的DH变形，但是不应该按照这种形式使用。在系统的其他部分使用之前，计算结果k需要进行Hash计算。更详细的讨论参见15.6节。

12.9 可能出错的地方

很少有书籍或文章谈到检查接收到的数是否位于正确的子群中的重要性。Niels首先在IPSec [41]的Internet密钥交换协议（IKE）中发现了这一问题。一些IKE协议包括了DH交换。

由于IKE协议需要在现实世界中操作，因此它必须处理丢失的消息。所以IKE详细说明了如果Bob没有收到应答，他应该重发最后一个消息。IKE没有指定Alice该如何处理Bob再次发送的消息。Alice容易犯下严重的错误。

为了简单起见，让我们假定Alice和Bob使用图12.3所示的子群中的DH协议，而不去检查X和Y是否为正确的值。此外，此次交换之后Alice开始使用新密钥k来发送加密并认证的消息给Bob，它包含了更多的协议数据（这是个很平常的情况，类似的情形在IKE中也会发生）。

Alice		**Bob**
已知：(p,q,g)		已知：(p,q,g)
检查(p,q,g)参数		检查(p,q,g)参数
$x \in_{\mathcal{R}} \{1,\ldots,q-1\}$		
	$\xrightarrow{X := g^x}$	
		$1 \overset{?}{<} X \overset{?}{<} p,\ X^q \overset{?}{=} 1$
		$y \in_{\mathcal{R}} \{1,\ldots,q-1\}$
	$\xleftarrow{Y := g^y}$	
$1 \overset{?}{<} Y \overset{?}{<} p,\ Y^q \overset{?}{=} 1$		
$k \leftarrow (Y)^x$		$k \leftarrow (X)^y$

图 12.3　子群中的Diffie-Hellman协议

Alice的危险行为说明如下：当她接收到包含Y的第二个消息的重发时，她只是重新计算密钥k并发送正确的应答给Bob。这听起来完全没有问题，对吗？但是攻击者Eve现在就可以做手脚。假设d是$(p-1)$的小因子。Eve可以把Y替换为一个阶数为d的元素。Alice的密钥k现在只限于d种可能，并且完全由Y和$(x \bmod d)$确定。Eve尝试$(x \bmod d)$的所有可能性，计算Alice可能得到的密钥k并试图解密Alice发送的下一个消息。如果Eve正确地猜到了$(x \bmod d)$，那么这个消息将正确地解密，从而Eve知道了$(x \bmod d)$。

然而，如果$p-1$包含了多个小因子$(d_1, d_2, \cdots, d_k)$怎么办？那么Eve就对每一个因子重复运行这个攻击，得到$(x \bmod d_1), \cdots, (x \bmod d_k)$。使用一般形式的中国剩余定理（见13.2节），她可以结合这些知识得到$(x \bmod d_1 d_2 d_3 \cdots d_k)$。所以，只要$p-1$的所有小因子的乘积比较大，Eve就得到了相当多的$x$的信息。由于$x$应该是保密的，这永远是一件坏事。在这个特定的情况下，Eve可以通过把原始的Y转发给Alice来结束，使得Alice和Bob能够完成该协议。然而，Eve已经收集到了足够的x的信息，从而可以马上找到Alice和Bob使用的密钥k。

说得更清楚一点：这不是对IKE的攻击。这是对标准允许的IKE实现的攻击。尽管如此，就我们看来，这个协议应该包括足够的信息，使得一个有能力的编程人员可以创建安全的实现。不考虑这些类型的信息是危险的，因为有人会在有些地方以错误的方式实现它。

Eve需要$p-1$碰巧有足够多的小因子。我们的设计要抵抗能够执行2^{128}步计算的敌手，这允许Eve利用$p-1$的所有大约小于2^{128}的因子。关于Eve能够得到多少信息，我没有看到过一个令人满意的概率分析，但是直接的估计表明，从小于2^{128}的因子中Eve平均能得到128比特的关于x的信息。然后她使用碰撞类型的攻击来攻击x的未知部分，并且由于x只有256比特，这就导致了一个真正的攻击。至少，当我们不去检查X和Y是否位于正确的子群中时，会发生这样的攻击。

如果Eve是选择子群(p, q, g)的人，那么攻击就变得容易了。可能在首先选择p的时候她自己已经把小因子放入$p-1$了。或者，也许她坐在标准委员会的委员旁边，推荐将某些参数作为标准。这并不像看起来的那么疯狂。美国政府以NIST的形式帮助提供用于DSA的素数。美国政府的其他机构（如NSA、CIA、FBI）是能够探听私人通信的特权机构。我们当然不希望这些素数是劣质的，但是这大概需要你在使用之前进行检查。这很容易做到。实际上，NIST公布了一个算法来选择不嵌入小因子的参数，你可以检查是否真的按照算法来做的。但是很少有人这样做过。

最后，最简单的办法是验证接收的每一个值都在正确的子群中。所有其他的防止小子群攻击的方法都要复杂很多。验证可以设法直接检测$p-1$的小因子，但是那种方法太复杂了。可以要求产生参数集合的人提供$p-1$的分解法，但那样会给整个系统增加许多复杂性。验证接收到的值位于正确的子群中只需要很少的工作，它是目前为止最简单、最有效的解决方案。

第13章　RSA

RSA系统大概是世界上最为广泛使用的公钥密码体制了。当然，它也是最有名的。它能够同时提供数字签名方案和公钥加密方案，从而使其成为一个通用的工具；它是基于大数分解的困难性而设计的，该问题在过去的几千年里迷住了很多人，并且得到了广泛的研究。

13.1　引言

RSA类似于Diffie-Hellman（见第12章），但二者有很大的区别。Diffie-Hellman（或者简称DH）是基于单项函数的：假定p和g是公开已知的，可以由x计算$(g^x (\bmod p))$，但是，给定$g^x \bmod p$不能计算出x。RSA是基于陷门单向函数的。给定公开已知的信息n和e，很容易由m计算$m^e \pmod n$，但相反的方向却不行。然而，如果知道n的分解，那么反向计算就变得容易了。n的分解就是陷门信息。如果知道它，就可以对函数求逆；如果不知道就不能求逆函数。这个陷门功能使得RSA既可以用于加密，又能够用于数字签名。RSA是由Ronald Rivest、Adi Shamir和Leonard Adleman发明的，首次发表于1978年[81]。

我们本章自始至终使用p, q和n。其中p和q是不同的大素数，每一个都有1000比特长或者更多。n定义为$n := pq$（普通的乘积，也就是说，没有模运算）。

13.2　中国剩余定理

我们将进行模合数n的计算，而不是像DH系统中的模素数p的计算。为了解释接下来要做什么，我们需要一些更多的关于模n计算的数论知识。一个非常有用的工具是中国剩余定理，或者简称为CRT。之所以如此命名，是因为其基本版本是由公元1世纪中国一位姓孙的数学家给出的。（关于DH和RSA所需要的大多数数学原理都可追溯到几千年前，所以它们不应该很难理解，对吗？）

模n的数有$0, 1, \cdots, n-1$。这些数不形成有限域，因为n不是素数。数学家仍然把这些数记为$\mathbb{Z}_n$并称之为环，但我们不需要这个术语。对于每一个$\mathbb{Z}_n$中的x，我们计算$(x \bmod p, x \bmod q)$。中国剩余定理表明，可以计算逆函数：如果知道$(x \bmod p, x \bmod q)$，就能够重构x。

为了便于标记，我们定义$(a, b) := (x \bmod p, x \bmod q)$。

我们首先说明重构是完全可能的，然后给出一个算法来计算原始的x。给定(a, b)，为了能够计算x，我们必须保证$\mathbb{Z}_n$中不会有第二个x'满足$x' \bmod p = a$和$x' \bmod p = b$。如果确实如此，那么x'和x都将导致相同的(a, b)对，而且没有算法能够断定这两个数中哪一个才是原始的输入。

PDG

设$d := x-x'$为产生相同的(a, b)对的两个数之差。那么$(d \bmod p) = (x-x') \bmod p = (x \bmod p) - (x' \bmod p) = a - a = 0$，因而$d$是$p$的倍数。同理，$d$是$q$的倍数。这意味着$d$是$\mathrm{lcm}(p, q)$的倍数，因为$\mathrm{lcm}(p, q)$是最小公倍数。因为$p$和$q$是不同的素数，$\mathrm{lcm}(p, q) = pq = n$，从而$x-x'$是$n$的倍数。但是，$x$和$x'$都在$0, \cdots, n-1$范围之内，所以$x-x'$一定是位于区间$-n+1, \cdots, n-1$之间的$n$的倍数。惟一的有效解是$x-x' = 0$，亦即$x = x'$。这就证明了对任意给定的$(a, b)$对，至多有一个解。我们所需要做的就是找到这个解。

13.2.1 Garner 引理

计算这个解的最有效的方法是Garner引理：

$$x = (((a-b)(q^{-1} \bmod p)) \bmod p) \cdot q + b$$

其中$(q^{-1} \bmod p)$是一个只依赖于p和q的常数。回想一下我们可以模p相除，因此能够计算$(1/q \bmod p)$，这不过是$(q^{-1} \bmod p)$的另一种写法。

不需要去理解Garner引理。我们需要做的是证明结果x是正确的。

首先，我们说明x位于正确的区间$0, \cdots, n-1$之内。显然，$x \geqslant 0$。而$t := (((a-b)(q^{-1} \bmod p)) \bmod p)$一定位于区间$0, \cdots, p-1$之内，因为它是模$p$的结果。如果$t \leqslant p-1$，那么$tq \leqslant (p-1)q$，从而$x = tq+b \leqslant (p-1)q + (q-1) = pq-1 = n-1$。这表明$x$在区间$0, \cdots, n-1$之内。

这个结果对于模p和模q来说都是正确的。

$$\begin{aligned}
x \bmod q &= ((((a-b)(q^{-1} \bmod p)) \bmod p) \cdot q + b) \bmod q \\
&= (K \cdot q + b) \bmod q & \text{对某一个 } K \text{ 成立} \\
&= b \bmod q \\
&= b
\end{aligned}$$

q前面所乘的全部内容是某一个整数K，但当进行模q计算时，q的任何倍数都是无关紧要的。模p有一点复杂：

$$\begin{aligned}
x \bmod p &= ((((a-b)(q^{-1} \bmod p)) \bmod p) \cdot q + b) \bmod p \\
&= (((a-b)q^{-1}) \cdot q + b) \bmod p \\
&= ((a-b)(q^{-1}q) + b) \bmod p \\
&= ((a-b) + b) \bmod p \\
&= a \bmod p \\
&= a
\end{aligned}$$

第一行只是扩展了$(x \bmod p)$，第二行除去了一对多余的$\bmod p$算子，然后交换了乘法的顺序，这不会改变结果（你可能还记得，在学校里学过，乘法是可结合的，所以$(ab)c = a(bc)$）。在接下来的一步中，我们注意到$q^{-1}q = 1 \pmod p$，于是可以把这一项完全去掉。其他的就是平凡的了。

这个推导过程比我们目前看到的都要复杂一些，尤其是使用了更多的代数性质。如果不能理解，也不用担心。

现在可以给出结论：Garner 公式给出了位于正确范围内的结果 x，且满足 $(a, b) = (x \bmod p, x \bmod q)$。鉴于我们已经知道只可能有一个这样的解，所以 Garner 公式完全解决了 CRT 问题。

在实际系统中，你可以预计算 $q^{-1} \bmod p$ 的值，于是 Garner 公式需要一次模 p 减法、一次模 p 乘法、一次完整的乘法以及一次加法。

13.2.2 推广

CRT 也适用于 n 是多个不同素数的乘积的情形①。Garner 公式也可以推广到这些情形中去，但本书不需要这样做。

13.2.3 应用

那么 CRT 有什么好处呢？如果要做大量的模 n 计算，那么使用 CRT 就节省了许多时间。对于一个数 $0 \leqslant x < n$，我们称 $(x \bmod p, x \bmod q)$ 对为 x 的 CRT 表示。如果有了 x 和 y 的 CRT 表示，那么 $x + y$ 的 CRT 表示就是 $((x + y) \bmod p, (x + y) \bmod q)$，这容易由 x 和 y 的 CRT 表示计算出来。第一个分量 $(x + y) \bmod p$ 可以计算为 $((x \bmod p) + (y \bmod p)) \bmod p)$。这正是每一个 CRT 表示的第一部分的和（模 p）。该结果的第二个分量能够以类似的方式计算。

乘法也可以用同样的方法计算。xy 的 CRT 表示是 $(xy \bmod p, xy \bmod q)$，很容易由各自的 CRT 表示计算得到。第一部分 $(xy \bmod p)$ 通过将 $(x \bmod p)$ 和 $(y \bmod p)$ 相乘并把所得结果再次模 p 而计算出来，第二部分可用同样的方式模 q 计算得出。

设 k 是 n 的比特数。每个素数 p 和 q 大约都是 $k/2$ 比特长。一次模 n 加法需要一次 k 比特的加法，如果其结果超过 n，接着需要一次 k 比特的减法。采用 CRT 表示需要对一半大小的数做两次模加法，其工作量大约是一样的。

对于乘法来说，CRT 节省了大量时间。两个 k 比特的数相乘所需要的工作量比两个 $k/2$ 比特的数相乘要多出两倍多。对于大多数实现，CRT 乘法比一个完整的乘法要快两倍，这种节省非常可观。

对于指数运算，CRT 能节省更多。假设要计算 $x^s \bmod n$，指数 s 可以达到 k 比特长。这大约需要 $3k/2$ 次模 n 乘法。利用 CRT 表示，每一次乘法的工作量要少一些，之外还有另外的节省。我们要计算 $(x^s \bmod p, x^s \bmod q)$。当模 p 计算时，可以把指数 s 模 $(p-1)$ 约化；对于模 q 也类似。因此只需要计算 $(x^{s \bmod (p-1)} \bmod p, x^{s \bmod (q-1)} \bmod q)$。这样，每一个指数只有 $k/2$ 比特长，只需要 $3k/4$ 次乘法。现在我们要做 $2 \cdot 3k/4 = 3k/2$ 次模其中一个素数的乘法，而不是 $3k/2$ 次模 n 的乘法。具体实现在计算时间上节省了 3 ~ 4 个因子。

① 有的版本适用于当 n 被某些素数的平方或者高次方幂整除的情况，但那些情况就更复杂了。

使用CRT惟一的代价是额外的软件复杂性和必要的转换。如果在一次计算中要做很多次乘法，而不是几次，这种转换的开销是值得的。大多数教科书只是提及CRT作为RSA的一种实现技巧。我们发现CRT表示使得RSA系统更易于理解，这就是我们先解释CRT的原因。稍后我们要利用它来解释RSA系统的性态。

13.2.4 结论

总之，当$n = pq$时，一个数x模n可以表示为$(x \bmod p, x \bmod q)$对。这两种表示之间的变换相当直接。如果要做很多次模一个合数的乘法，而你又知道该合数的分解，CRT表示是有用的（如果不知道n的分解，就不能用它来加速计算）。

13.3 模n乘法

在深入研究RSA的详细资料之前，我们必须考虑模n的数是如何进行乘法运算的。这与以前讨论的模p情形有一些不同。

对任何素数p，我们知道方程$x^{p-1} = 1 (\bmod p)$对所有的$0 < x < p$都成立。若模为合数n，这就不正确了。为了使RSA能够工作，我们需要找到一个指数t使得$x^t = 1 \bmod n$对（几乎）所有的x都成立。很多教科书都给出了答案，但对于读者理解为什么该答案是正确的并没有帮助。实际上，使用CRT可以很容易地找到正确的答案。

我们想要找到一个t，使得$x^t = 1\ (\bmod n)$对所有的x都成立。上面这个方程意味着$x^t = 1 (\bmod p)$和$x^t = 1 (\bmod q)$。由于p和q都是素数，这只有当$p-1$是t的因子且$q-1$是t的因子时才成立。所以具有这个性质的最小的t就是$\mathrm{lcm}(p-1, q-1) = (p-1)(q-1)/\gcd(p-1, q-1)$。本章的其余部分约定：$t = \mathrm{lcm}(p-1, q-1)$。

每一个人都在使用字母p、q和n，其中有人使用的是大写字母。大多数书不使用这个t，而是使用Euler ϕ函数$\phi(n)$。对形如$n = pq$的数n，其Euler ϕ函数可以计算为$\phi(n) = (p-1)(q-1)$，这是t的倍数。$x^{\phi(n)} = 1$当然是正确的，使用$\phi(n)$替代t也可得到正确的结果，但是使用t则更精确。

在我们的讨论中跳过了一个小问题：如果$x \bmod p = 0$，那么$x^t \bmod p$不可能等于1。所以方程$x^t \bmod n = 1$不可能对所有的x值都成立。不会有很多的数有这种不足。有q个数满足$x \bmod p = 0$，p个数满足$x \bmod q = 0$，于是所有存在问题的值的个数是$p+q$。或者更确切地说是$p+q-1$，因为我们把0值计算了两次。对于全部的值的个数$n = pq$，这一部分是可忽略的。更好的是，实际上RSA使用的性质是$x^{t+1} = x (\bmod n)$，该性质即使对这些特殊的数也成立。此外，当使用CRT表示时也容易看出这一点。如果$x = 0 (\bmod p)$，那么$x^{t+1} = 0 = x (\bmod p)$，这对于模$q$也同样成立。于是基本性质$x^{t+1} = x (\bmod n)$被保持下来，并且对于$Z_n$中所有的数都成立。

13.4 RSA 详细说明

下面将对 RSA 系统进行详细说明。首先随机选择两个不同的大素数 p 和 q，计算 $n = pq$。素数 p 和 q 应该是（差不多）同样的长度，结果模 n 的长度是 p 和 q 的两倍。

我们使用两个不同的指数，通常称为 e 和 d。对 e 和 d 的要求是 $ed = 1(\text{mod } t)$，其中 $t := \text{lcm}(p-1, q-1)$。公开指数 e 应该选取为某一个小的奇数值，而使用 11.3.5 节的 extendedGCD 函数计算作为 e 模 t 的逆的 d。这保证了 $ed = 1(\text{mod } t)$。

要加密消息 m，发送方计算密文 $c := m^e(\text{mod } n)$。为了解密密文 c，接收方计算 $c^d(\text{mod } n)$。这等于$(m^e)^d = m^{ed} = m^{kt+1} = (m^t)^k \cdot m = (1)^k \cdot m = m(\text{mod } n)$，其中 k 是某个存在的值。所以接收方能够解密密文 m^e 而得到明文 m。

(n, e)对构成了公钥，这些公钥分发给很多不同的参与方。数值(p, q, t, d)是私钥，由产生 RSA 密钥的人秘密保存。

为了方便起见，我们常常写为 $c^{1/e} \text{ mod } n$，而不是 $c^d \text{ mod } n$。模 n 计算的指数都要取模 t 运算，因为 $x^t = 1(\text{mod } n)$，所以指数 t 的倍数不会影响结果。而我们计算的 d 是 e 模 t 的逆，所以把 d 记为 $1/e$ 是自然的。记号 $c^{1/e}$ 常常更容易理解，尤其是使用多个 RSA 密钥的时候。这就是我们也说取一个数的 e 次根的原因。再回忆一下，计算模 n 的任何次根都需要掌握私钥。

13.4.1 使用 RSA 进行数字签名

迄今为止，我们只讨论了使用 RSA 来加密消息。RSA 的一个巨大优势在于既可以用于加密消息，又能够对消息签名。这两个操作使用相同的计算。为了签署一个消息 m，私钥的拥有者计算 $s := m^{1/e} \text{ mod } n$。现在$(m, s)$就是一个带签名的消息。要验证签名，任何知道公钥的人都能够检验 $s^e = m(\text{mod } n)$。

同加密一样，签名的安全性基于这样的事实：m 的 e 次根只有知道私钥的人才能够计算。

13.4.2 公开指数

到目前为止，我们描述的程序有一个问题：如果 e 与 $t = \text{lcm}(p-1, q-1)$有公因子，那么 d 就没有解，所以必须选取 p、q 和 e 使得这种情形不会发生。这更像是一件麻烦事，而不是一个问题，但是必须要处理。

选取一个小的公开指数使得 RSA 更高效，因为计算一个数的 e 次幂所需要的计算量更小了。因此我们设法为 e 选择一个小的值。本书中我们将为 e 选择一个固定值，并选择 p 和 q 使其满足上述条件。

你必须很小心，以使加密函数和数字签名函数不会以不希望的方式互相影响。你不希望一个攻击者可能通过说服私钥的拥有者签署 c 来解密一个消息 c。毕竟，签署“消息” c 与解密密文 c 是相同的操作。本书后面给出的编码函数将防止出现这种情况，但我们仍然不希望对两个

函数都使用相同的操作。可以对加密和认证使用不同的RSA密钥，但是这会增加复杂度，而且密钥的数量也会加倍。

我们的解决办法是，对同一个n使用两个不同的公开指数。对于签名使用$e=3$，对于加密使用$e=5$，这就消除了系统之间的相互影响，因为模n的立方根和五次根是互相独立的，知道其中的一个不会帮助攻击者计算另一个[29]。

选择固定的e值简化了系统，也带来了可预见的性能。它确实也对所使用的素数增加了限制，因为$p-1$和$q-1$都不能是3或者5的倍数，这在生成素数的第一步很容易检验。

使用3和5的根本原因很简单，它们是最小的合适的值[①]。我们选取了小的公开指数用于签名，因为签名通常要多次验证，而任意一块数据只需要加密一次，因此使签名验证操作更高效是很有意义的。

其他常用的e值有17和65 537。我们喜欢小的数值，因为它们更高效。使用小的公开指数存在一些潜在的小问题，但是我们将使用编码函数进一步消除该问题。

使用小的d值也不错，但是这里不得不让你失望了。尽管找到一对具有小d值的(e, d)是可能的，但是使用小d是不安全的[95]，因此不要敷衍塞责而为d选择一个方便的值。

13.4.3 私钥

如果攻击者只知道公钥(n, e)，寻找私钥p、q、t或者d中的任何一个值都是极其困难的。只要n足够大，没有算法会在可接受的时间内做到这一点。我们知道的最好的方法是把n分解为p和q，然后从中计算t和d。这就是为什么你常常听到因子分解对密码学是如此重要的原因。

我们已经谈到，私钥由p、q、t和d组成。事实上，知道这其中的任何一个值都足以计算出所有其他3个值。这是非常有启发意义的。

我们假定攻击者知道公钥(n, e)，因为这是公开的信息。如果他知道p或者q，事情就容易了。给定p就能够计算$q = n/p$，然后计算t和d，就像我们上面所做的那样。

如果攻击者知道了(n, e, t)会怎么样呢？首先，$t = (p-1)(q-1)/\gcd(p-1, q-1)$，但是由于$(p-1)(q-1)$很接近于$n$，寻找$\gcd(p-1, q-1)$比较容易，因为它是最接近于$n/t$的整数［$\gcd(p-1, q-1)$的值绝不会很大，因为两个随机数不大可能共享一个大的因子］，这使得攻击者能够计算$(p-1)(q-1)$。他也能够计算$n-(p-1)(q-1)+1 = pq-(pq-p-q+1)+1 = p+q$。所以现在他有$n = pq$和$s := p+q$，因此可以推导出下面的方程：

$$
\begin{aligned}
s &= p + q \\
s &= p + n/p \\
ps &= p^2 + n \\
0 &= p^2 - ps + n
\end{aligned}
$$

① 理论上可以使用$e=2$，但是那会带来许多额外的复杂性

最后一个方程只是一个关于p的二次方程，攻击者能够使用中学数学进行求解。当然，一旦攻击者知道了p，他就可以计算出所有其他的私钥。

如果攻击者知道了d，同样的事情也会发生。在我们的所有系统中，e是非常小的。由于$d < t$，数$ed-1$只是一个小因子乘以t。攻击者只需猜测这个因子，计算t，然后试图按照上面的方法找到p和q。如果失败他就尝试其他可能的因子。（有一些更快的技巧，但这个是比较容易理解的一个。）

简而言之，知道p、q、t或者d中的任何一个值，都使得攻击者能够计算出所有其他的值。所以假定私钥的拥有者秘密保存这4个值将是安全的。在实现中只需要存储其中的一个，但是人们常常存储所需要的几个值来执行RSA解密操作。这是依赖于实现的，从密码学的角度来看这是无关的。

如果Alice想要解密或者签发一个消息，显然她必须知道d。因为知道d等价于知道p和q，可以安全地假定她知道n的因子，因而可以使用CRT表示来进行计算。这样太好了，因为计算一个数的d次幂是RSA中最耗时的运算，而使用CRT表示就节省了3～4倍的工作量。

13.4.4 *n*的长度

模n应该与DH情形中的模p具有相同的长度，详细讨论见12.7节。需要重申的是：如果要保护数据达20年之久，n的绝对最小长度是2048比特。这个最小值将随着计算机速度变得越来越快而缓慢增长。可能的话，在应用中就取n为4096比特长，或者接近于这个长度。此外，要确信你的软件支持直到8192比特长的n值。你不会知道将来会怎样，如果你能够使用大的密钥而不用更换软件或者硬件，它将会发挥作用。

两个素数p和q应该具有相同的长度。要得到一个k比特的n，可以生成两个随机的$k/2$比特的素数并把它们相乘。你可能得到的是一个$k-1$比特的模n，但那也没有多大关系。

13.4.5 产生RSA密钥

为了把所有的事情结合在一起，我们给出两种方法来产生具有期望的性质的RSA密钥。第一种方法是对11.4节中generateLargePrime函数的修改。惟一的功能变化是我们要求素数满足$p \bmod 3 \neq 1$和$p \bmod 5 \neq 1$，以保证使用公开指数3和5。当然，如果要使用一个不同的固定e值，必须相应地修改这个方法。

function GenerateRSAPrime
input: k 期望的素数长度，按比特数计。
output: p 位于$2^{k-1}, \cdots, 2^k-1$之间的素数，满足$p \bmod 3 \neq 1 \wedge p \bmod 5 \neq 1$。
　敏感范围检查。
　assert $1024 \leqslant k \leqslant 4096$
　计算最大的尝试次数。
　　$r \leftarrow 100k$

```
repeat
  r ← r−1
  assert r > 0
  选择一个随机的 k 比特数。
  n ∈ R 2^(k−1), …, 2^k−1
  继续尝试直至找到一个素数。
until n mod 3 ≠ 1 ∧ n mod 5 ≠ 1 ∧ isPrime(n)
return n
```

我们仅仅指定了素数的长度，而没有指定素数应该落入的完整区间。这是一个很具灵活性的较小的定义，虽然有些简单，但对RSA来说足够了。额外的要求在循环条件中。如果n不是合适的模3或者模5的数，明智的实现将不调用isPrime(n)，因为isPrime要占用相当数量的计算。

那么，为什么还要在循环计数器中包括错误条件呢？的确，既然区间足够大，我们总会找到一个合适的素数吗？希望如此，但奇怪的事情发生了。我们不担心所取的区间中没有素数，我们担心的是一个拙劣的PRNG，它总是返回相同的结果。遗憾的是，这是随机数发生器的一个常见的错误模式，这个简单的检查使得generateRSAPrime是安全的，不会造成PRNG失常。另一个可能的错误模式在于拙劣的isPrime总是判断一个数是合数。

下一个函数将产生所有的密钥参数。

```
function generateRSAKey
input:  k     期望的素数长度，按比特数计。
output: p, q  模的因子。
        n     大约为 k 比特的模。
        d_3   签名指数。
        d_5   解密指数。
敏感范围检查。
assert 2048 ≤ k ≤ 8192
产生素数。
p ← generateRSAPrime(⌊k/2⌋)
q ← generateRSAPrime(⌊k/2⌋)
一个小的检测，万一 PRNG 是坏的……
assert p ≠ q
计算 t 为 lcm(p−1, q−1)
t ← (p−1)(q−1)/GCD(p−1, q−1)
利用扩展 GCD 算法的模逆特性计算私钥指数。
g, (u, v) ← extendedGCD(t, 3)
```

检查 GCD 是否正确，或者根本就没有得到逆元。

assert $g = 1$

模 t 约化 u，因为 u 可能是负的，而 d_3 不应该是负的。

$d_3 \leftarrow u \bmod t$

现在是 d_5。

$g, (u, v) \leftarrow \text{extendedGCD}(t, 5)$

assert $g = 1$

$d_5 \leftarrow u \bmod t$

return p, q, pq, d_3, d_5

注意，我们已经选取了固定的公开指数，并产生了既能够用于签名（$e = 3$）又能够用于加密（$e = 5$）的密钥。

13.5 使用 RSA 的缺陷

使用目前给出的RSA是很危险的，问题在于其数学结构。比如，如果Alice数字签发了两个消息 m_1 和 m_2，那么 Bob 能够计算 Alice 关于 $m_3 := m_1m_2 \bmod n$ 的签名。要知道，Alice 计算了 $m_1^{1/e}$ 和 $m_2^{1/e}$，Bob 可以把二者相乘从而得到$(m_1m_2)^{1/e}$。

如果Bob使用Alice的公钥加密了一个很小的消息 m，就会产生另一个问题。如果 $e = 5$ 而 $m < \sqrt[5]{n}$，那么 $m^e = m^5 < n$，于是没有模约化发生。攻击者 Eve 仅仅通过计算 m^5 的五次根就可以恢复 m，这是容易做到的，因为不涉及模约化。按照这种方式产生错误的典型情形就是 Bob 设法发送 AES 密钥给 Alice。如果他只是把 256 比特的值作为一个整数，那么加密后的密钥不超过 $2^{256 \cdot 5} = 2^{1280}$，这比我们的 n 小得多。所以绝不会有模约化，Eve 只需要计算加密后的密钥值的五次根就得到了密钥。

我们如此详细地解释RSA背后的理论，原因之一在于要讲授我们所遇到的一些数学结构。正是上面的数学结构引起了很多种攻击。最简单的一个攻击已经在前面提到了。有一些更高级的攻击是基于求解模n的多项式方程技巧的。所有这些都传递了一个信息：RSA所操作的数可以有任何类型的结构，这是非常糟糕的。

解决方法是利用一个函数来破坏所有的可用结构。有时这称为填充函数，但是这个词用得不当。填充一词通常用于增加字节以得到具有正确长度的结果。对于 RSA 加密和签名，人们使用了各种形式的填充方案，只有少数的设计方案导致了攻击。你所需要的是一个能够尽可能去掉结构的函数，我们称之为编码函数。

关于这个问题有一些标准，其中最著名的是 PKCS #1 v2.1[85]。同样，这不是一个专一的标准。有两个 RSA 加密方案和两个 RSA 签名方案，每一个都可以选取不同的 Hash 函数。这未必是坏事，但我们不想增加额外的复杂性。所以我们给出一些更简单的方法，即使它们可能不具有一些 PKCS 方法的所有特性。

PKCS #1 v2.1 标准也说明了一个技术文档编制的普遍问题：它混淆了规范与实现。RSA 解密函数制定了两次，一次使用了方程 $m = c^d \bmod n$，一次使用了 CRT 方程。这两次计算的结果是一样的：一个仅仅是另一个的优化实现。如此的实现描述不应该成为标准的一部分，因为它们不会产生不同的性态，因此应该分开讨论。我们不想专门评论这个 PKCS 标准，这是一个非常普遍的问题，在计算机行业里到处可见。

13.6 加密

加密消息是 RSA 的规范应用，但在实际中几乎从不使用。原因很简单：使用 RSA 所能够加密的消息的长度限定于 n 的长度。在真实系统中，你甚至不能使用所有这些比特，因为编码函数是有开销的。有限的消息长度对大多数应用来说太不实际了，而且由于从计算的角度来看 RSA 操作相当耗时，因此不希望把一个消息分成小块而使用单独的 RSA 操作来加密每一块消息。

到处使用的解决方案几乎都是选择一个随机的密钥 K，利用 RSA 密钥加密 K，然后使用密钥 K 通过分组密码或者流密码来加密实际的消息 m。所以是发送 $E_{RSA}(K), E_K(m)$，而不是发送 $E_{RSA}(m)$。这样，消息的长度不再有限制，而且即使对于很大的消息也只需要一次 RSA 操作。当然，必须传送少量的额外数据，但是与得到的好处相比，这个代价一般是很小的。

我们可以使用一个更为简单的加密方法。作为选择一个 K 并加密 K 的替代，我们选择一个随机数 $r \in \mathbb{Z}_n$ 并定义块加密密钥为 $K := h(r)$，其中 h 是某一个 Hash 函数。对 r 的加密只是计算它的模 n 的五次幂（记住，对于加密我们使用了 $e = 5$）。这个方法简单而安全。因为 r 是随机选择的，r 没有结构可以用于攻击加密方案的 RSA 部分。而 Hash 函数则保证了不同的 r 的结构不会拓展到 K 的结构，除了相同输入必须产生相同的输出这个明显的必要条件之外。

为了实现的简单化，我们在 $0, \cdots, 2^k-1$ 区间选择 r，其中 k 是满足 $2^k < n$ 的最大的整数。产生一个随机的 k 比特数要比产生 $\mathbb{Z}_n$ 中的一个随机数容易一些，而且分布一致的微小偏离在这种情形下是无害的。

下面是更形式化的定义。

function encryptRandomKeyWithRSA
input: (n, e) RSA 公钥，在我们这里 $e = 5$。
output: K 被加密的对称密钥。
c RSA 密文。

计算 k。
$k \leftarrow \lfloor \log_2 n \rfloor$
选择随机数 r，满足 $0 \leq r < 2^k-1$。
$r \in R\ \{0, \cdots, 2^k-1\}$
$K \leftarrow \text{SHA}_d\text{-256}(r)$

$c \leftarrow r^e \bmod n$
return (K, c)

接收方计算 $K = h(c^{1/e} \bmod n)$，得到同一个密钥 K。

function decryptRandomKeyWithRSA
input: (n, d) 对应于 $e = 5$ 的 RSA 私钥。
c 密文。
output: K 被加密的对称密钥。
assert $0 \leqslant c < n$
这是显然的。
$K \leftarrow \text{SHA}_d\text{-}256(c^{1/e} \bmod n)$
return K

我们已经详细地讨论了在给定私钥的情况下如何计算 $c^{1/e}$，所以这里就不再讨论了。只是不要忘了使用 CRT 可使速度提高 3~4 倍。

下面有一个考虑安全性的好方法。假设 Bob 为 Alice 加密了密钥 K，而 Eve 想知道有关密钥的更多知识。Bob 的消息只依赖于某一个随机的数据和 Alice 的公钥。于是，最坏的情况是这个消息可以向 Eve 泄露有关 K 的数据，但是不会泄露任何其他的秘密数据，如 Alice 的私钥。密钥 K 是由 Hash 函数计算得到的，而我们可以认为 Hash 函数是一个随机的映射（如果不能把 Hash 函数看做一个随机的映射，它就不满足我们关于 Hash 函数的安全需求）。得到有关 Hash 函数的输出信息的惟一途径是知道其输入。这意味着要掌握 r 的信息。但是，如果 RSA 是安全的——而我们必须这样假定，因为必须要使用它——那么，只给定 $(r^e \bmod n)$ 就不可能得到随机选取的 r 的大量信息，这就给攻击者留下了关于 r 的大量的不确定性，从而得不到 K 的知识。

假设密钥 K 在以后暴露给了 Eve，这可能是因为系统其他部分出了故障。这是否会暴露 Alice 的私钥的一些信息呢？不会。K 是 Hash 函数的输出，Eve 不可能从 Hash 函数中推导出任何有关输入的信息。所以即使 Eve 以某种特殊的方式来选择 c，她所得到的 K 也不会暴露任何有关 r 的信息。Alice 的私钥只是用于计算 r，所以 Eve 也不会获得有关 Alice 私钥的任何知识。

这是在 decryptRandomKeyWithRSA 函数中使用 Hash 函数的一个好处。假如它仅输出 $c^{1/e} \bmod n$，那么这个程序就可以用于发动各种攻击。假设系统的某一部分有弱点，Eve 得知了输出的最低有效位，那么 Eve 就可以发送特殊选取的值 $c_1, c_2, c_3, \cdots$ 给 Alice，得到 $c_1^{1/e}, c_2^{1/e}, c_3^{1/e}, \cdots$ 的最低有效位。这些应答有各种各样的代数性质，Eve 很可能从这种情形中获得一些有用的信息。decryptRandomKeyWithRSA 中的 Hash 函数 h 破坏了所有的数学结构。从输出 K 中获得一

个比特，Eve几乎得不到任何有关$c^{1/e}$的信息，即使整个结果K也只是暴露很少的有用信息。Hash函数是不可逆的。增加Hash函数使得RSA程序在抗击系统其他故障时更加安全。

这也是为什么decryptRandomKeyWithRSA不检查我们由c计算出的r是否位于区间$0, \cdots, 2^k-1$之内的原因。如果我们检查了这个条件，就必须处理可能产生的错误。错误处理总是会导致不同的行为，而Eve很可能会探测到是否有错误发生了。这将向Eve提供暴露信息的机会：Eve能够选择任何值c并获知是否$c^{1/e} \bmod n < 2^k$。没有Alice的帮助，Eve不能够计算这个性质。如果能够避免的话，我们不希望帮助Eve。不检查该条件，我们至多会产生一个无用的输出，而这看起来在任何情况下都是会发生的，因为c可能被误用而不会产生一个有效的r值[①]。

说一点儿题外话：暴露一个随机$(c, c^{1/e})$对与对他人选择的c计算$c^{1/e}$之间是有很大差别的。任何人都能够产生形如$(c, c^{1/e})$的数对。你所做的只是选择随机的r，然后令$c := r^e$，并计算(r^e, r)对。这些数对没有什么秘密可言。然而，如果Alice非常友好地为从Eve那里接收到的c计算$c^{1/e}$，Eve可以选择具有某些特殊性质的c值，而Eve自己不能产生这个$(c, c^{1/e})$数对。不要为你的攻击者提供这种额外的服务。

13.7 签名

对于签名，我们必须多做一点儿工作。问题在于要签发的消息m可以有许多结构，我们不希望这些数具有任何结构，因为对这些数我们要计算RSA根，因此必须破坏这些结构。

第一步是对这些消息进行Hash运算。所以，我们处理的是固定长度的值$h(m)$（其中h是Hash函数），而不是处理可变长度的消息m。如果使用SHA_d-256，我们可得到256比特的结果。但是n比这大得多，所以不能直接使用$h(m)$。

简单的办法是使用一个伪随机映射把$h(m)$扩展成位于$0, \cdots, n-1$范围内的一个随机数，然后计算m的签名为$s^{1/e} (\bmod n)$。把$h(m)$映射为一个模n的数需要一些工作量（参见10.8节的讨论）。在这个特殊的情形中，我们可以安全地把这个问题简化：把$h(m)$映射为一个位于$0, \cdots, 2^k-1$范围之内的随机的元素，其中k是满足$2^k < n$的最大的整数。位于$0, \cdots, 2^k-1$之内的数很容易产生，因为我们只需要产生k个随机比特。在这个特定的情形下，这是一个安全的办法，但是不要到处使用它。在密码学中，很多情形下这样做会破坏整个系统。

我们将使用第10章给出的Fortuna PRNG的生成器。为此，很多系统使用Hash函数来构造特殊的小随机数发生器，但我们已经定义了一个好的发生器。此外，需要PRNG来选择素数以产生RSA密钥，所以你的软件中已经有PRNG了。

这样就产生了3个函数——一个把消息映射为s，一个签发消息，而第三个用于验证签名。

① 对r增加更多的限制也不会阻止无意义输出的问题。Eve总是可以使用Alice的公钥以及修正的encryptRandomKeyWithRSA函数发送给Alice一些无意义密钥的加密。

function MsgToRSANumber
input: n RSA 公钥，模。
m 消息，将转换为一个模 n 的值。
output: s 一个模 n 的数。
创建一个新的 PRNG 发生器。
$g \leftarrow$ InitialiseGenerator( )
使用消息的 Hash 作为种子。
ReSeed(g, SHA$_d$-256(m))
计算 k。
$k \leftarrow \lfloor \log 2\ n \rfloor$
$x \leftarrow$ GenerateRandomData(g, $\lceil k/8 \rceil$)
与往常一样，我们把比特串 x 看做整数，使用 LSByte 的第一个约定。模约化可以通过对 x 的最后字节进行简单的 AND 操作来实现。
$s \leftarrow x \bmod 2^k$
return s

function SignWithRSA
input: (n,d) 对应于 $e=3$ 的 RSA 私钥。
m 待签发的消息。
output: σ m 的签名。
$s \leftarrow$ MsgToRSANumber(n, m)
$\sigma \leftarrow s^{1/e} \bmod n$
return σ

小写字母 σ 又称 sigma，经常用于表示签名，因为它在希腊语中等价于字母 s。到如今你该知道给定私钥该如何计算 $s^{1/e} \bmod n$ 了。

function VerifyRSASignature
input: (n, e) RSA 公钥，其中 $e=3$。
m 待要签发的消息。
σ 消息的签名。
$s \leftarrow$ MsgToRSANumber(n, m)
assert $s = \sigma^e \bmod n$

当然，在实际应用中，如果签名验证失败则要采取一定的措施。我们这里只是写下了一个断言以表明正常的操作该终止了。签名的失败应该和密码协议的其他错误一样处理：作为一

个明确的信号表示你正受到主动攻击。除非不得已，不要发送任何应答，而且要销毁所有计算的东西。发送出去的信息越多，给攻击者的信息也越多。

关于RSA签名的安全性讨论与RSA加密很类似。如果要求Alice签发一串消息$m_1, m_2, \cdots, m_i$，那么得到的是形如$(s, s^{1/e})$的数对，但是s值实际上是随机的。只要Hash函数是安全的，只能通过反复试验来冒充$h(m)$。此外，随机数发生器也是随机的映射，任何人都可以生成形如$(s, s^{1/e})$的数对，其中s值是随机的，所以没有提供任何新的信息以帮助敌手来伪造签名。但是，对任何特定的消息m，只有知道私钥的人才能够计算相应的$(s, s^{1/e})$对，因为s必须由$h(m)$计算出来，然后$s^{1/e}$必须由s计算得到。这就需要私钥。因此，任何验证签名的人都知道Alice一定签发了该消息。

这样我们就结束了RSA的处理，也结束了本书数学味道很浓厚的章节。我们将把DH和RSA应用于密钥协商协议和PKI，但只会用到我们解释过的数学知识，因此不会再介绍新的数学知识了。

第 14 章　密码协商协议介绍

密码协议由参与方之间的消息交换组成。我们在第 12 章中已经看到了一个简单的密码协议。

协议大概是密码学中最困难的部分了。最主要的问题在于，作为设计者或者实现者，协议不在你的控制之下。到目前为止，我们一直在设计系统并控制不同参与方的行为。一旦开始与其他参与方进行通信，就控制不了他们的行为。其他参与方有与你不同的兴趣范围，他可以背离协议规则而设法获得一些利益。当设计协议时，必须假定你是在与敌手打交道。

14.1　角色

协议一般描述为由 Alice 和 Bob 执行，或者在客户和商家之间执行。像"Alice"、"Bob"、"客户"、"商家"这些名字并不真正表示一个特定的个体或者机构，他们代表协议中的一个角色。如果 Smith 先生要和 Jones 先生进行通信，他可能要执行一次密钥协商协议。Smith 先生可以扮演 Alice 的角色，而 Jones 先生则扮演 Bob 的角色。第二天他们的角色可能会互换。有一点很重要，谨记单个实体可以扮演任何角色[①]。在分析密码协议的安全性时，记住这一点尤为重要。我们已经在 DH 协议中看到了中间相遇攻击。在那个攻击中，Eve 同时扮演了 Alice 和 Bob 的角色（当然，Eve 也是另外一个角色）。

14.2　信任

信任是我们与他人进行交往的最根本的基础。如果你根本不信任任何人，为什么要与他们打交道呢？比如，购买单独包装的块状糖需要最基层的信任。客户需要信任商家会提供糖果并正确地找回零钱。商家需要信任客户会付款。如果对方不按规则行事，双方都可以求助于法律。在商店偷窃商品的人是要受到起诉的。有欺骗行为的商家也要冒承担坏名声、法律诉讼、被指责等风险。

下面是几种类型的信任。

道德规范　道德规范对社会有很大的影响。人们依照伦理行事，大部分人绝大多数时间都是遵循道德规范的。攻击者是少数的。大多数人买东西都会付账，即使在偷东西很容易的情况下。

① 在具有三个或者更多参与方的协议中，甚至一个人同时扮演多个角色也是可能的。

名誉 在我们的社会中，有一个“好名声”是非常重要的。个人和公司都想要保护他们的名誉。背负坏名声的威胁通常是他们正确行事的动机。

法律 在文明社会里，有一个法律基础设施支持对行为不端的人进行诉讼和检举。这也是人们正确行事的一个动机。

身体威胁 另一个正确行事的动机是害怕一旦因欺骗被抓到后遭受伤害。这是针对毒品交易和其他非法贸易的信任来源之一。这种威胁可以是身体暴力或者其他行为。

MAD 这是一个冷战术语：全称是Mutually Assured Destruction。往轻了说，这是一种既伤害自己又伤害对方的威胁。如果欺骗了你的朋友，她可能会断绝你们的关系，给你们都带来伤害。有时你看到两个公司处于MAD情形，尤其是在他们就侵犯专利权互相提起诉讼时。

所有这些信任的来源都是参与方有非欺骗动机的机制。另一方知道这个动机因而感觉他可以在一定程度上信任对手。这就是为何与完全失去理性的人交易时这些动机都会失败的原因：你不能信任他们会根据其最大利益行事，从而破坏了所有这些机制。

在因特网上取得信任是困难的。假设Alice生活在国外并连接到ACME网站。对于我们提到的信任机制，ACME几乎没有任何理由信任Alice，剩下的只有道德规范了。海外私人法律援助几乎是不可能的，而且毫无疑问费用是惊人的昂贵。

Alice和ACME之间还存在一种信任基础，因为ACME要保护其声誉。在设计电子商务协议时记住这一点很重要。如果存在任何的失效方式（而这总是存在的），这种失效应该是由于ACME的原因，因为ACME有通过人工干预来正确解决问题的动机[①]。如果失效是由于Alice的原因，那么这个问题不大可能得到正确解决。此外，ACME将容易受到攻击，攻击者试图导致错误模式并从中获利。

信任问题不是黑和白的问题。不是说你要么信任某个人，要么就不信任他。对不同的人有不同的信任等级。你可能对一个朋友的信任度是100美元，而不是500万美元。我们信任银行会保护我们的存款安全，但却需索要收据和作废支票的副本，因为我们不是完全信任他们的管理。“你信任他吗？”这个问题是不完整的。应该是“在*X*这件事情上你信任他吗？”

14.2.1 风险

信任是商业交易的基本原则，但是通常它以风险的形式表示，而不是以信任的方式表示。风险看起来正好与信任相反。风险能够以多种形式评估、比较、交易。

当涉及到密码协议时，谈论信任比谈论风险更容易些。但缺乏信任是一种风险，有时这可以通过标准的风险管理技巧来处理，比如保险。当设计协议时，我们谈论的是信任。你要

① 几乎所有的个人电话、邮件、电子商务都遵循这个规则，在货物运送之前客户要为其订单付款。

始终谨记，交易的人们总是根据风险进行思考和谈论。如果想同他们会谈，必须在这两种观点之间进行转换。

14.3　动机

动机结构是协议分析的另一个基本的组成部分。不同参与者的目的是什么？他们想要得到什么？即使在现实生活中，分析动机结构也会给出富于洞察力的结论。

有很多次，每周我们都得到一些新闻报道，宣称“新的研究表明……”。我们的第一反应总是要问：谁资助了该研究？其结果有利于资助方的研究总是受到人们的怀疑。这里给出几个因素。首先，研究人员知道他们的客户想要听什么，而且知道如果他们能提出“好”的结果就可以再次得到合同。这就会引起偏见。其次，研究工作的赞助者不打算公布任何负面报告。烟草公司发表“科学”报告来说明烟碱不会上瘾，微软公司资助研究来“证明”公开源代码软件在某些方面是有害的。不要总是信任那些被赞助公司的研究工作。

作者本人非常熟悉这些压力。在多年的顾问生涯中，我们为赞助客户完成了很多次安全性评估。我们经常很严厉——评估的一般产品非常糟糕——对其组件我们很少给予肯定的评价。这样做并不总是受到客户的欢迎。其中一个客户曾经给 Bruce 打电话说：“停止你们的工作并把票据发给我，我已经找到要价更低的人了，他们会写出更好的报告。”猜测一下这里的“更好”指的是什么？能够做到公正的惟一原因是我们有足够的工作。如果工作很少而且不得不为生计而工作，这种诱惑是很大的，你就会忍不住说一些客户希望听到的话。

在其他领域中我们也能看到同样的问题。就在我们写这本书的时候，新闻报道到处都是关于会计师和银行业的故事。分析家和会计师为其客户撰写他们喜欢的报告，而不是公正的评估。我们要谴责激励结构，它们为这些人的报告带有偏见提供了理由。着眼于这些动机是非常有益的，我们为此已经工作了多年。有了一点实践经验就很容易了，并且会得出有价值的见解。是的，它使你对人们的动机产生愤世嫉俗的看法。

如果你请人管理职工优先认股权，就给他们这样的动机结构：在接下来的 3 年内提高每股的价格就赚大钱，减少每股的价格也可以拿到解雇金。这属于“正面向上就赚大钱，背面朝上赚小钱”的激励，猜猜有些管理者会怎么做？他们会努力获取高风险的短期策略。如果有机会把股价翻倍，他们一定会放手一博，因为只会赢利而不会有损失。如果通过账目上的欺骗手段可以在几年内使股价提升，他们会这样做的，因为在被发现之前他们就可以宣布破产。

类似的情况在 20 世纪 80 年代就发生于美国的储蓄和贷款业。联邦政府放开了限制，允许金融联合会自由投资。与此同时，政府向他们担保储蓄。我们来看看这个动机结构。如果投资赢利了，那么金融联合会就获利，管理人员无疑也会得到巨额奖金。如果投资失利了，联邦政府会为存款人支付存款。丝毫也不奇怪，一帮金融联合会在高风险的投资中损失了大量的资金——于是联邦政府接管了他们的账单。

确定动机结构通常是相对容易的。比如，股票交易所可以安排和支付账目的审计，而不是公司自己来支付审计。如果对审计师发现的每一个错误都给予巨额奖金，那么就可以得到更加精确的审计报告。

现实中存在着大量不合理的动机结构的例子。负责离婚诉讼的律师的动机非常尖刻，因为他们会在为个人财产辩护的每一小时得到报酬。如果诉讼费用超过了个人财产的总价值，我敢打赌他们会建议你尽快解决这一问题。

在美国，法律诉讼是很普通的。如果发生了意外事故，每一个当事人都有掩盖、否认的巨大动机，或者以其他的方式来逃避责任。乍一看来，严厉的法律责任和巨额的伤害赔偿金对社会是有益的，但是它极大地阻碍了人们判断事故发生的原因，从而难以确定在以后如何来避免类似事故的发生。倾向于保护消费者的责任法几乎不可能使得像Firestone这样的公司承认他们的产品存在问题，所以我们都要学会采取更好的办法。

密码协议与动机结构的相互作用体现在两个方面。首先，它们依赖于动机结构。一些电子支付协议没有阻止商家欺骗消费者，但是却为消费者提供了受欺骗的证明。这样做是必要的，因为商家有动机不让消费者拿出他们进行欺骗的证据。这些证据可以用于法庭诉讼，或者是有损商家的声誉。

密码协议也改变了动机结构，它们使得某些事情变得不可能，从而在动机结构中消除了这些事情。它们也带来了新的可能性和新的动机。一旦你使用了网上银行，那就给小偷创造了入侵计算机窃取钱财的动机。

起先，动机看起来主要是唯物论的，但这仅仅是一部分。许多人具有非实物主义的目的。多数计算机入侵并不打算得到什么东西，而仅仅是开玩笑、获得特权或者吹牛的资本。在人际关系上，绝大多数的动机与钱财没有什么关系。保持一个清醒的头脑，尽力理解人们的动机，然后据此创造协议。

14.4 密码协议中的信任

密码协议的功能是为了将需要的信任减到最少。让我们重复一遍，密码协议的功能是为了将需要的信任减到最少。这意味着同时最小化需要彼此信任的人数和他们需要的信任量。

设计密码协议的一个强大工具是偏执狂模型。当Alice参与一个协议时，她假定所有其他的参与方共同合谋来欺骗她。这是真正的最终合谋理论。当然，其他每个参与方也做了同样的假设，这是所有密码协议设计的默认模型。

对于这个默认模型的任何背离都必须有明确的证明。很奇怪的是，这一步常常被忽略。比如，多数Web站点使用了SSL协议。SSL协议需要可信的证书。浏览器经常接受各种证书，而一个证书是很容易得到的。结果是，用户在安全地与一个Web站点通信时并不知道是在与哪一个Web站点通信。针对PayPal用户的众多诈骗都利用了这个弱点。

不用证明一个特定协议所需的信任是很诱人的，因为它常常是“显然的”。对于协议的设计者来说这可能是真实的，但是如同系统的任何模块一样，按照惯例，协议应该有一个明确的详细说明界面。

从商业的角度来看，备有文件证明的信任需求也列出了各种风险。所需信任的每一点都暗示了一种必须处理的风险。

14.5　消息和步骤

典型的协议描述包括一些在协议的参与方之间传送的消息数量和每个参与方需要进行的计算的描述。

几乎所有的协议都在很高的层面进行描述，大多数细节都没有被描写。这虽然有助于你关注协议的核心功能，但也造成了很大的危险。没有仔细描述每个参与方应该做的所有动作，要建立安全的协议实现是及其困难的。

有时你会看到协议定义了所有的细节和校验。这样的说明书通常非常复杂，没有人能够完全理解它们。这也许有助于实现者，但是太复杂的东西不可能是安全的。

解决方法是模块化。如同通信协议一样，要使用密码协议，可以把需要的功能分割成几个协议层。每一层都在前一层的基础上工作。所有这些层都是重要的，但对于所有协议来说，大多数层都是一样的。只有最高层是高度可变的，这是总能找到备有证明文件的那一个。

14.5.1　传输层

网络专家一定要原谅我们在这里重复使用他们的术语。对我们这些密码学家来说，传输层就是允许参与方进行通信的根本通信系统。它由一个参与方发送给另一个参与方的字节串组成。如何做到这一点就与我们的目的无关了。作为密码学家，我们关心的是一个参与方可以给另一个参与方发送字节串，可以使用 UDP 包、TCP 数据流、E-mail 或者任何其他方法。很多情况下，传输层需要额外的编码。比如，如果一个程序同时执行多个协议，传输层必须把消息递送给正确的协议执行。这可能需要某种额外的目的地字段。使用 TCP 时，需要包含消息的长度，用以在流定向的 TCP 协议中提供消息定向服务。

为了清楚起见，我们期望传输层能传送任意的字节串。在一个消息中任意的字节值都可能出现。串的长度是可变的。接收到的串当然应该与发送的是一样的，删除末尾的零字节或者任何其他的修改都是不允许的。

有的传输层包括了像魔术常量之类的东西，用于及早检测到错误或者检查TCP流的同步。如果接收到的消息的魔术常量不正确，其他的消息就应该丢掉。

有一个重要的特殊情况：有时我们在一个密码保护的信道上运行密码协议，就如在第 8 章中设计的那样。在这样的情形下，传输层也提供了机密性、认证以及重放保护等特性。这就使得协议更容易设计，因为我们担心的攻击类型少了很多。

14.5.2 协议和消息标识

下一层（向上的）提供协议和消息的标识。当接收到一个消息时，一定希望知道它属于哪一个协议，并且是该协议中的哪一个消息。

协议标识符通常由两部分组成。第一部分是验证信息，它为未来升级提供了空间；第二部分标识了该消息属于哪一个特定的密码协议。在电子支付系统中，可能是提款协议、支付协议、存款协议、退款协议等。协议标识符避免了在不同协议的消息之间产生混淆。

消息标识符指出了它是该协议中的哪一个消息。如果协议中有4个消息，你一定不希望有任何混淆。

为什么要包括这么多识别信息呢？攻击者不能伪造它们吗？他当然可以。这一层不提供保护以防止伪造攻击；相反，它检测意外错误。意外错误的良好检测是很重要的。假设你负责维护一个系统，突然之间得到了大量的错误信息。区分主动攻击和诸如配置及版本问题的意外错误是颇有价值的服务。

协议和消息标识符也使得消息更加独立，这使得大量的维护和调试更加容易。汽车和飞机的设计都易于维护。软件就更复杂了——有更多的理由说明其设计应该易于维护。

包括消息识别信息的最重要的原因大概与Horton原理有关。在协议中使用认证（或者数字签名）时，我们通常认证几个消息和数据字段。通过包含消息识别信息可以避免把消息解释为错误内容的风险。

14.5.3 消息编码和解析

下一层是编码层。消息的每一个数据元素都必须转换成一列字节。这是一个标准的编程问题，这里不打算就此讨论太多的细节。

一个很重要的问题是解析。接收方必须能把看起来像一个字节串的消息解析回它的要素字段。这个解析不能依赖于上下文的信息。

在协议的所有版本中相同的固定长度的字段是容易解析的。你清楚地知道它有多长。当一个字段的长度或者意义依赖于某些上下文信息时，比如协议的早期消息，问题就出现了。这是比较麻烦的事。

密码协议中许多消息的尾端都带有签名的信息或者认证的信息。认证函数认证了一个字符串，而通常最简单的办法是在传输层来认证消息。如果拦截到的消息依赖于某些上下文信息，那么签名或者认证是不明确的。根据这一类型的错误，我们攻破了几个协议。

编码字段的一个好方法是使用Tag-Length-Value或者TLV编码方案。每一字段都编码为3个数据元。Tag标识了目前处理的字段，Length标识值编码的长度，而Value则是待编码的实际数据。流传久远的TLV编码方式是ASN.1 [43]，但是它有着非常糟糕的复杂的说明，我们应避免使用它。而ASN.1的一个子集合则是非常有用的。

一个新的选项是XML。忽略XML的类型，我们只使用XML作为一个数据编码系统。只要使用了固定的文档模板定义（DTD），那么解析就不依赖于上下文，并且不会有任何问题。

14.5.4　协议执行状态

在许多实现中，单个计算机有可能同时参与几个协议的执行。为了跟踪所有的协议，需要某种形式的协议执行状态。这个状态包括完成协议所需要的所有信息。

实现协议需要某种事件驱动的编程，因为这些执行在继续进行之前必须等待外部消息的到达。这可以有几种方式来实现，比如使用单线程或采用每一个协议的执行方式，或者使用某种类型的事件分派系统。

给定了事件驱动编程的基本构造，实现协议就相当直接了。协议状态包括了一个状态机，它表明了下一个期待的消息的类型。作为一般的规则，没有其他类型的消息是可接受的。如果所期待类型的消息到了，就将其解析并根据规则进行处理。

14.5.5　错误

协议总要包括多重检查。这些检查包括检验协议类型及消息类型，检查它是否为协议执行状态所期待类型的消息，然后解析消息，并执行它所指定的密码验证。如果其中的任何一个检验失败，我们就会遇到一个错误。

错误需要非常小心地处理，因为它们是潜在的攻击手段。最安全的方法是：对于错误不发送任何回应并立即删除协议状态。这样最小化了攻击者能够得到的关于协议的信息量。遗憾的是，这样就会产生一个不友好的系统，因为它没有任何错误指示。

为了使系统是可用的，通常需要增加某些类型的错误消息。如果能够避免的话，就不要给协议中的其他参与方发送错误消息。在安全日志上记录下错误消息以便系统管理员可以诊断该问题。如果必须发送错误消息，尽可能使得它不提供情报。诸如“出现错误”之类的简单消息通常就足够了。

一个危险的交互作用来自于错误和计时攻击之间。攻击者Eve可以发送一个伪造的消息给Alice并等待她的错误回答。Alice察觉错误并发送回答的时间常常包含了具体的信息，表明什么出错了以及哪个地方出错了。

下面是一个充分的例子说明了这些交互作用的危险。几年前，Niels从事一个商用的智能卡系统的工作。其中一个性质是需要PIN代码来激活智能卡。将四位数的PIN代码发送给智能卡，它会回应一个消息以表明智能卡现在是否已经被激活。该实现大约需要做10 000次尝试来穷举所有可能的PIN代码。智能卡允许5次失败的尝试，然后就锁上了，之后需要其他特殊的方式才能解锁。这一想法允许不知道PIN代码的攻击者可以做5次尝试来猜测这个四位数的PIN代码，那么在卡锁上之前，她就有1/2000的可能性猜测到PIN代码。

设计是完好的，同样的设计如今被广泛使用。而1/2000的机会对许多应用来说已经足够好了。但是很不幸，个别的智能卡系统编程人员有一点儿粗心。要验证四位数的PIN码，程序首先检查第一位数，然后是第二位，依次类推。一旦它检查到其中的一位出错了，智能卡马上报告PIN码出现错误。其弱点就在于：智能卡发送“错误的PIN”所需要的时间依赖于有几位PIN数字是正确的。聪明的攻击者可以测量到这个时间从而得到大量的信息。特别值得注意的是，攻击者能够发现第一个出错的数位在哪一个位置上。具备了这样的知识，攻击者只需要40次尝试就可以穷举搜索整个PIN空间（10次尝试之后第一位数字一定是正确的，再有10次尝试第二位数字也是正确的，依次类推）。经过5次尝试之后，她找到正确的PIN码的机会就增加到1/143，这比她应该具备的1/2000的机会大多了。如果能得到20次尝试，她的机会就增加到60%，这比她应该具有的0.2%的概率大很多。

更糟糕的是：在某些情况下进行20次到40次尝试并非不可行。在几次失败的尝试之后会锁上的智能卡，一旦使用了正确的PIN就会重置计数器，于是用户得到了另外5次尝试机会以用于下次输入正确的PIN。假设你的室友有一个类似上面所述的智能卡。如果能拿到室友的智能卡，你可以做一到两次尝试，然后把它放回去。等待他真的在某个地方使用了智能卡，利用了正确的PIN并且在智能卡中把失败的PIN尝试计数器重置，那么就可以再做一到两次尝试。很快你就可以得到整个PIN码，因为它至多需要40次尝试就可以发现。

错误的处理是很复杂的，很难给出一些简单的规则。这方面我们知道的尚不够多。目前，我们只能说：“要非常小心”。

14.5.6 重放和重试

重放攻击发生于这样的情形：攻击者记录了一条消息，然后再次发送同样的消息。我们必须要保护系统免受重放攻击。要察觉到这种攻击可能有一些棘手，因为消息看起来的确像是正确的。毕竟，它就是一个正确的消息。

与重放攻击紧密相关的是重试攻击。假设Alice正在与Bob执行一个协议，并且她没有得到应答。这可能有多种原因，但是最常见的原因是Bob没有收到Alice的最后一条消息并一直在等那条消息。在现实生活中这总是会发生的，而我们解决这个问题的办法是发送另外一封信或者E-mail，或者重复我们的最后一句话。在自动化的系统中，这称为重试。Alice重试她的最后一条消息给Bob并再次等待回答。

于是，Bob就收到了攻击者发送的消息重放，并重新发送消息给Alice。Bob要设法正确地处理这些消息以保证正确的行为，从而不带来安全缺点。

发送重试消息相当简单。每一个参与方都有某种形式的协议执行状态。你需要做的是保持一个计数器，如果在相应的时间内没有收到答复，就再次发送最后一条消息。确切的时间界限依赖于所使用的通信基础设施。如果使用UDP包（一个直接使用IP包的协议），存在一个消息

会丢失的合理概率，于是你希望有一个短的重试时间，大概是几秒钟的样子。如果使用TCP来发送消息，那么TCP会使用它自己的时间间隔重新发送任何没有被正确接收的数据。在密码协议层，没有多少理由需要重试，大多使用TCP的系统都不会这样做。不过，在下面的讨论中我们打算假定使用重试，因为一般的处理接收重试的技巧同样适用，即使你从没有发送它们。

当接收到一个消息时，必须判断如何来处理它。我们假设每一条消息都是可识别的，以便知道它应该是协议中的哪一条消息。如果它正是你所期盼的消息，则只需要遵从协议的规则执行就可以了；假如它是一条协议中的“将来”的消息，比如正好是在以后的某个时刻你期盼的消息。这也很容易：不理睬它。不要改变你的状态，不要发送答复，只需要丢掉它然后什么也不做。这可能是攻击的一部分。即使在可能是由于消息丢失而导致的一系列错误的协议中，忽略这个消息与消息在传输中丢失有同样的效果。由于假定协议会恢复丢失的消息，所以不理睬消息总是一个安全的办法。

剩下来的情形就是“旧”的消息：在协议运行中已经处理过的消息。在三种情况下这会发生这样的事情。第一种情况：接收到的消息是以前响应过的，它与你响应的消息是一样的。在这种情况下，该消息很可能就是一次重试，所以发送在第一次响应时发送的同样的回答即可。注意，回答一定要是同一个。不要使用不同的随机值重新计算回答，也不要简单地假定你所计算得到的应答消息与第一次回答的是一样的。你必须进行检查。

第二种情况：你接收到一个消息，它与你上次响应的消息具有同样的消息标识符，但是消息的内容却不一样。比如，假设在DH协议中Bob收到了来自Alice的第一条消息，而后来又收到了另外一条消息，声称是协议的第一条消息，但是包含了不同的数据。这种情况是攻击的预示。任何重试都不可能出现这种情形，因为重新发送的消息绝不可能与第一次发送的不一样。要么你刚刚收到的消息是假的，要么你早些时候响应的消息是伪造的。安全的选择是把它看做一个协议错误，并按照我们讨论过的来做（忽略你刚刚收到的消息是安全的，但也同样意味着发现了少数形式的主动攻击。这样做对于安全系统的检测和响应有一些不利的影响）。

第三种情况：你接收到了一个消息，它甚至比你响应的前一个消息更旧。对此你不需要做什么。如果你还知道在协议的那个阶段所接收到的原始消息，就检查它们是否一样。假如的确一样，就不要理睬它；如果不一样，那么就检测到了一个攻击，应该把它当做一个协议错误。很多实现并不存储协议执行中接收到的所有的消息，从而不可能分辨现在接收的消息是否与原先处理过的消息是一样的。安全的选项是忽略这些消息。你可能对经常发生这些事情感到吃惊。有时消息会延迟一段时间。假设Alice发送的消息延迟了，几秒钟之后她发送重试，并且的确到达了，然后Alice和Bob都继续执行协议。半分钟之后，Bob接收到了最初的消息。在这种情形中，按照协议的术语，我们说Bob接收到了很旧的消息的复制。

如果在一个协议中不止有两个参与方，那么事情就变得更复杂了。这些情形是存在的，但是超出了本书的范围。如果你曾经从事一个多方协议的工作，一定要仔细考虑重放和重试。

最后给出一个评论：你不可能知道协议的最后一条消息是否已经到达。如果Alice发送了最后一条消息，那么她绝不可能得到消息已经到达的确认。如果通信链接被破坏，而Bob从没有收到最后一条消息，那么Bob重试前一条消息，但是Alice同样也收不到。从协议的正常终止来看，Alice是不能区分这些情况的。可以让Bob增加一条致谢消息，在协议的最后发送给Alice，但是，这个致谢变成了新的最后消息，同样的问题又出现了。密码协议的设计方式必须确保这种不确定性不会导致不安全的行为。

第 15 章　密钥协商协议

最后，我们来讨论密钥协商协议，这种协议的目的是获取一个共享密钥，而此共享密钥用于建立我们在第 8 章定义的安全通道。

完整的密钥协商协议是十分复杂的，把最终协议一次性全部表述出来可能会引起混淆。于是，我们把它描述成一系列的协议，每个协议都增加了一点儿功能。必须清楚的是，这些中间协议不完全实用，它可能有很多弱点。

15.1 背景

这个协议有两个参与方：Alice 和 Bob。Alice 和 Bob 想进行安全的通信。他们首先要执行密钥协商协议来建立一个秘密的会话密钥 k，然后使用会话密钥 k 建立一个安全通道进行实际数据的交换。

对于一个安全的密钥协商过程来说，Alice 和 Bob 必须能够识别对方的身份。这种基本的认证能力是本书第三部分讨论的主要内容。而现在，我们仅假设 Alice 和 Bob 能够认证彼此的信息，这种基本的认证可以通过使用 RSA 签名（如果 Alice 和 Bob 知道对方的密钥或使用 PKI）或使用一个共享密钥和一个 MAC 函数来实现。

但是先等一等！如果你已经有了一个共享的密钥，为什么还要执行密钥协商协议呢？其中有很多原因需要你这样做。首先，密钥协商能够消除会话密钥与已经存在的（长期）共享密钥之间的相互影响。如果会话密钥泄露了（例如，因为有缺陷的安全通道的实现），而共享秘密仍然是安全的。同时，如果密钥协商协议运行之后共享密钥泄露了，即使攻击者知道共享的密钥仍然不能获取由该协议协商出来的会话密钥。因此，如果你今天丢失了密钥，昨天的数据仍然是被保护的。这些是重要的性质：它们使整个系统具有更强的健壮性。

在某些情况下，共享的密钥是相对脆弱的，比如口令。用户不喜欢记住一个长达 30 个字符的口令，而是倾向于去选择简单的口令。一个标准的攻击就是字典攻击，其中计算机搜寻大量的简单口令。一个好的密钥协商协议能够把一个脆弱的口令转变成强健的密钥。然而，在本章中我们不考虑这些更高级的协议。

15.2 初试

我们以所能想到的最简单的设计开始，见图 15.1。这正是附加了认证的一个子群中的 DH 协议。Alice 和 Bob 通过首先发送的两条信息来执行 DH 协议（为了简单起见，我们放弃了一

些必需的检验）。Alice 随后对会话密钥 k 计算一个认证值，并把它传送给 Bob，由 Bob 来检验这个认证值。同样，Bob 也发送密钥 k 的一个认证值给 Alice。

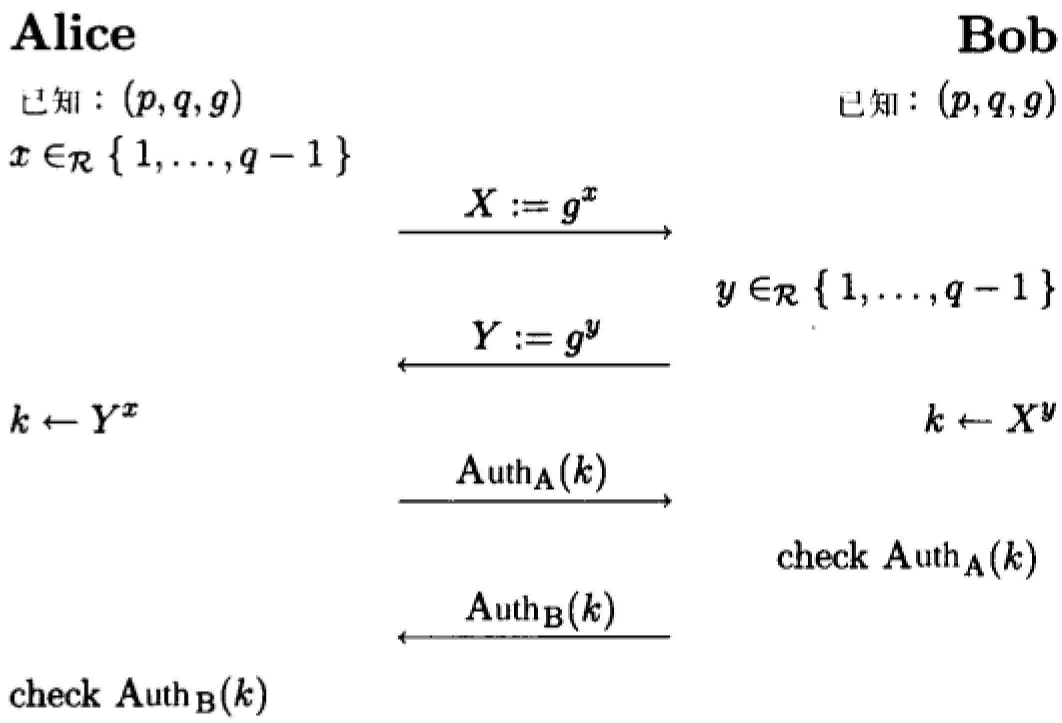


图 15.1 协商密钥的第一次尝试

此时，我们并不知道这些认证值的确切形式。必须记住的是，我们说过假设 Alice 和 Bob 能认证彼此的消息。因此，Bob 能够检测（check）$\mathrm{Auth}_{\mathrm{A}}(k)$，同时 Alice 也能够检测到 $\mathrm{Auth}_{\mathrm{B}}(k)$。关于使用数字签名还是使用 MAC 函数，不是我们所关心的。这种协议只是将认证能力转化为一个会话密钥。

该协议中存在以下几个问题：

- 这个协议假设 Alice 和 Bob 都知道 (p, q, g)。把这些值设置为常量是一个很糟糕的主意。
- 协议使用 4 条消息，然而只使用 3 条消息就可能达到目的了。
- 会话密钥被用做认证函数的输入。如果认证函数是强健的，这不是一个问题，但是假如认证函数泄露了会话密钥一些比特位，那么情况会变得很糟糕，因此当然需要对整个协议进行新的分析。根据经验，比较好的原则是一个密钥仅用于一件事情。在这里，k 将用来作为会话密钥，因此，我们不希望把它用做认证函数的输入变量。
- 协议中的两条认证消息太相似了。例如，如果认证函数是简单的 MAC，所使用的密钥是 Alice 和 Bob 都知道的，那么 Bob 可以简单地发送他从 Alice 那里得到的认证值，从而他不需要密钥就可以完成整个协议。因此，Alice 将不能够确信最后一条认证信息。
- 实现过程中必须小心，除非认证消息已经交换，否则不能使用密钥 k。这不是一个大问题，而是一个很简单的要求，但是，当某些程序员试图优化一个程序时，你想像不到他们会发明出什么来。

我们将在本章的讲解中解决所有这些问题。

15.3 永久性的协议

我们已经强调了系统抵抗未来攻击的重要性。对协议来说，这一点更为重要。如果把数据库字段的大小限定为2000个字节，对有些用户来说可能会受到影响，但是可以在下一版本中取消这一限制。对协议来说情况就不一样了。协议在不同的参与者之间运行，每一个新的版本需要和老的版本之间实现互操作。修改一个协议并且保持它和老的版本的兼容是相当复杂的工作。在了解这些之前，你不得不在系统上执行各种版本的协议，以便决定该使用哪个版本的协议。

当然，协议版本的变换将成为一个攻击点。如果旧的协议不太安全，攻击者会对强迫你使用旧版本协议很感兴趣。你可能会对我们看到的系统遭受这种称做版本回退的攻击之多感到吃惊。

要知道未来的所有需求当然是不可能的，因此，在某个时候及时定义协议的第二个版本是有必要的。然而，协议存在几个版本的代价是很高的，尤其在整体的复杂性方面。

一个成功协议的生命周期是永久性的（我们不关心那些不成功的协议）。从世界上完全除去一个协议是及其困难的。因此，设计能够经得起未来检验的协议就更加重要。这就是为什么我们不能为密钥协商协议指定一组固定的DH参数的原因。即使我们选择很大的参数，仍然存在危险：未来密码技术的进步可能迫使我们改变这些参数。

15.4 认证约定

在进一步阐述之前，我们介绍一种认证约定。通常，协议包含很多不同的数据元素，而且很难断定哪些数据元素需要进行认证。一些协议被攻破就是因为忽略了对某些数据字段的认证。我们采用一种简单的约定来解决这些问题。

在我们的协议中，参与方每次发送一个认证消息，认证数据包含目前所有交换的数据——所有以前的消息和在认证消息之前认证者消息中的所有数据字段。在图 15.1 所示的协议中，Alice 的认证信息不在 k 上，而是在 X 和 Y 上，Bob 的认证消息则包含了 X、Y 和 Auth_A。

这种约定阻止了大量的攻击方法，代价也很低。密码协议并不交换如此多的数据，同时认证计算几乎总是由对输入的字符串进行Hash计算开始。Hash函数运算速度非常之快，以至可以忽略额外的代价。

这种约定也使得我们可以缩短记号，我们可以简单地记为 Auth_A，而不是记做诸如 $\mathrm{Auth}_A(X, Y)$之类的形式。由于待认证的数据由这一约定指定，因此我们不再需要精确地记录下来。本书中所有其他的协议都将使用这种约定。

提示：认证函数只是认证一个字符串。每一个待认证的字符串必须以惟一的标识符开头，以标识协议中用于认证的确切位置。同时，早先的消息和数据字段转换成字符串的编码方式，

必须使这些消息和字段可从字符串中恢复，而无需其他的上下文信息。我们对此已经进行了很详细的讨论，但这也是我们容易忽略的一个要点。

15.5 第二次尝试

我们如何解决前一个协议的问题呢？我们不想使用一个不变的DH参数集，因此，我们让Alice去选择参数，并发送给Bob。

如图15.2所示，我们把4条消息压缩为两条。Alice由选择DH参数和她自己的DH贡献值开始，连同认证值一起发送给Bob。Bob必须检查DH参数是否是正确选取的、X是否是有效的（检查的详细信息参见第12章）。协议的其余部分和前一个版本相似。Alice接收到Y和Auth_B，检查它们，并计算DH结果。

Alice		Bob
Choose suitable (p,q,g)		
$x \in_{\mathcal{R}} \{1,\ldots,q-1\}$		
	(p,q,g), $X := g^x$, Auth_A $\longrightarrow$	
		Check (p,q,g), X, Auth_A
		$y \in_{\mathcal{R}} \{1,\ldots,q-1\}$
	$\longleftarrow$ $Y := g^y$, Auth_B	
Check Y, Auth_B		
$k \leftarrow Y^x$		$k \leftarrow X^y$

图15.2 协商密钥的第二次尝试

我们不再确定DH参数。我们只用到了两条消息，没有以任何方式直接使用认证密钥，并且认证约定确保了待认证的字符串是不相同的。

但是现在我们面临以下一些新的问题：

- 如果Bob想要一个比Alice更大的DH素数，我们该如何做呢？或许Bob有更为严格的安全策略，并且认为Alice选择的DH参数的安全性不够。Bob将不得不终止这一协议。或许他会在“Require DH prime to be at least k bits long”（“需要的DH素数至少为k比特长”）后面发送一条错误信息，但这样会变得很麻烦和复杂。Alice则必须选择新的参数来重新启动这个协议。
- 认证也存在一个问题：Bob根本不能确定自己是在和Alice交谈。任何人都能记录Alice发送的第一条消息，并在以后发送给Bob。Bob则认为该消息来自Alice（毕竟，认证通过了检验），从而完成协议并认为他与Alice共享了一个密钥k。由于不知道x，攻击者不能获得k；而没有密钥k，攻击者不能侵入系统中使用k的其余部分。但是Bob的日志

会显示与Alice进行的一次完整的认证协议，这就是问题本身，因为它为调查的管理员提供了错误信息。

Bob的问题称为缺乏“活性”（“liveness”）。他不能确信Alice是否“在线”，并且他不是在和重放的幽灵交谈。解决这一问题的传统方法是确信Alice的认证值包括了Bob选取的一个随机元素。

15.6　第三次尝试

让我们再做一些改变来解决这些问题。这一次Alice不选取DH参数，而只是将她的最低要求发送给Bob，由Bob来选取DH参数。这样做把消息的数量增加到3条（事实上，大多数有趣的密码协议至少需要3条消息。对此，我们并不知道其中的原因，他们就是这样做的）。Bob仅发送一条消息，即第二条消息。该消息包括他的认证值，因此，Alice应该在第一条消息中发送一个随机选取的元素。这里，我们使用一个随机数。

这样产生的协议如图15.3所示。Alice首先选择她想使用的素数p的最小长度s。她还要选择一个随机的256比特长的字符串作为随机数N_a并把它们都发送给Bob。Bob选取一个合适的DH参数集和他自己的随机指数，并把DH参数、他自己的贡献值以及他的认证值发送给Alice。Alice和往常一样，使用另一个认证来完成DH协议。

Alice		**Bob**
$s \leftarrow \min p$ size $N_a \in_{\mathcal{R}} 0, \ldots, 2^{256}-1$		
	$\xrightarrow{s,\ N_a}$	
		Choose (p, q, g) $x \in_{\mathcal{R}} \{1, \ldots, q-1\}$
	$(p, q, g),\ X := g^x,$ $\xleftarrow{\text{Auth}_B}$	
Check (p, q, g), X, Auth_B $y \in_{\mathcal{R}} \{1, \ldots, q-1\}$		
	$\xrightarrow{Y := g^y,\ \text{Auth}_A}$	
		Check Y, Auth_A
$k \leftarrow X^y$		$k \leftarrow Y^x$

图 15.3　协商密钥的第三次尝试

还有另外一个问题需要解决。因为最终结果k是一个长度可变的数，因而系统的其他部分会感到很难做到协调一致。此外，k是利用代数关系计算出来的，在密码系统中留下代数结构总是让我们感到恐慌。在一些地方必须采用这种结构，但我们应尽可能地避免它。

代数结构的威胁在于攻击者可能会发现某种方法来求解它。数学是一个非常强大的工具。在过去的20年中，我们看到了许多新的公钥密码系统提议几乎都被攻破了，这主要是由于其中包含的代数结构所致。只要可以，总是要除去任何代数结构。

最明显的解决办法是对最终的密钥进行Hash运算。这会使它缩减到固定的长度，而且也破坏了其中仍然存在的代数结构。

15.7 最终协议

最终协议的简化形式如图15.4所示，这是一个最易于阅读和理解的形式。然而，我们略去了协议中大量的验证步骤，以使其易于阅读。我们简单地记做“Check (p, q, g)”来表示几个验证步骤。为了显示所有要求的密码检验，图15.5给出了完整形式的协议。

Alice		**Bob**
$s \leftarrow \min p$ size $N_a \in_{\mathcal{R}} 0, \ldots, 2^{256}-1$		
	$\xrightarrow{s,\ N_a}$	
		Choose (p, q, g) $x \in_{\mathcal{R}} \{1, \ldots, q-1\}$
	$(p, q, g),\ X := g^x,$ Auth$_\text{B}$ $\longleftarrow$	
Check (p, q, g), X, Auth$_\text{B}$ $y \in_{\mathcal{R}} \{1, \ldots, q-1\}$		
	$\xrightarrow{Y := g^y,\ \text{Auth}_\text{A}}$	
		Check Y, Auth$_\text{A}$
$k \leftarrow \text{SHA}_d\text{-}256(X^y)$		$k \leftarrow \text{SHA}_d\text{-}256(Y^x)$

图15.4 最终协议的简化形式

Bob需要选择一个适当长度的素数p，这依赖于Alice所需的最小长度和他自己需要的最小长度。当然，Bob应确保s值的合理性。我们不希望Bob生成一个100 000比特的素数，因为他收到了含有较大s值的未经认证的消息。同样，Alice不应当仅仅因为Bob已经发送给她了就开始检查非常大的素数。因此，Alice和Bob都要限制p的长度。使用固定的最大值限制了灵活性。如果密码学突然有了很大的进展从而迫使我们使用大的素数，那么固定的最大值将真会成为一个问题。几乎没有人理解可配置的最大值带来的所有参数配置问题。我们已经选择使用一个动态的最大值。Alice和Bob都拒绝使用比他们推荐使用的素数大两倍的素数。动态的

最大值提供了很好的升级渠道，同时避免了超大素数的使用。我们可以讨论因数2是否是最好的。或许你会采用因数3，这没有太大关系。

Alice		**Bob**
$s_a \leftarrow \min p$ size $N_a \in_{\mathcal{R}} 0, \ldots, 2^{256}-1$		
	$\xrightarrow{s_a,\ N_a}$	
		$s_b \leftarrow \min p$ size $s \leftarrow \max(s_a, s_b)$ $s \overset{?}{\leqslant} 2 \cdot s_b$ Choose (p, q, g) with $\log_2 p \geqslant s - 1$ $x \in_{\mathcal{R}} \{1, \ldots, q-1\}$
	$(p,q,g),\ X := g^x,$ Auth$_{\mathrm{B}}$ $\longleftarrow$	
Check Auth$_{\mathrm{B}}$ $s_a - 1 \overset{?}{\leqslant} \log_2 p \overset{?}{\leqslant} 2 \cdot s_a$ $255 \overset{?}{\leqslant} \log_2 q \overset{?}{\leqslant} 256$ Check p, q both prime $q \overset{?}{\mid} (p-1) \wedge g \overset{?}{\neq} 1 \wedge g^q \overset{?}{=} 1$ $X \overset{?}{\neq} 1 \wedge X^q \overset{?}{=} 1$ $y \in_{\mathcal{R}} \{1, \ldots, q-1\}$		
	$\xrightarrow{Y := g^y,\ \mathrm{Auth_A}}$	
		Check Auth$_{\mathrm{A}}$ $Y \overset{?}{\neq} 1 \wedge Y^q \overset{?}{=} 1$
$k \leftarrow \mathrm{SHA}_d\text{-}256(X^y)$		$k \leftarrow \mathrm{SHA}_d\text{-}256(Y^x)$

图 15.5　长形式的最终协议

协议的其他部分只是前面的简化形式的扩展。如果 Alice 和 Bob 够精明的话，他们会使用合适的 DH 参数缓存。这将使 Bob 不需要每次都生成一个新的 DH 参数，同时也使 Alice 不必每一次都进行检查。应用程序甚至可以使用一个固定的DH参数集，那样就不必明确地传送它们。单一的 DH 参数集标识符就足够了。所有这些变化是对协议的简单而直接的优化，因此，我们不再进行详细讨论了。但是，一定要小心，优化可能导致足够多的协议修改以至最终被攻破。没有简单的规则去检测一种优化方式是否可以攻破协议。协议的设计与其说是一门科学还不如说是一门艺术，因此，没有严格的规则可以遵循。

15.8 看待协议的不同观点

有很多有益的方法来考虑这一协议。协议应该具备一些特性，我们能看到为什么协议能够提供这些特性。

15.8.1 Alice 的观点

让我们从 Alice 的观点来查看这一协议。她从 Bob 那里接收到单一的消息。因为消息是已认证的，并且这个认证消息包括了她的随机值 N_a，所以 Alice 确信这个消息来自 Bob。其他人没有任何方法可以伪造这一消息或者重放旧的消息。

Alice 检验这些 DH 参数是否是正确选取的，以使 DH 协议具有她所期望的所有性质。因此，当她对 y 保密而发送 Y 的时候，她确信只有知道满足 $g^x = X$ 中 x 的人才能计算出最终的密钥 k。这就是基本的 DH 协议的性质。Bob 认证 X，只有当他依照协议执行步骤操作时才这样做。于是，Bob 知道正确的 x 并对其保密。因此，Alice 确信只有 Bob 知道她推导出的最终密钥 k。

于是Alice相信自己确实正在与Bob进行交谈，并且确信她所获得的密钥只有她自己和Bob知道。

15.8.2 Bob 的观点

现在让我们看一下 Bob 的立场。他所接收到的第一条消息几乎没有给他什么有用的信息，它主要表述的是某个人已经选取一个值 s_a 和某些随机比特 N_a。

第三条消息则是不同的。这条消息一定来自 Alice，因为 Alice 认证了这一消息，并且我们一开始就假定了 Bob 能验证 Alice 的认证。这一认证包括 X 和 Bob 选取的一个随机值，所以第三条消息不是一个重放，而是 Alice 为这次协议的运行而特别认证的。与此同时，Alice 的认证包括了 Bob 接收到的第一条消息，所以他现在知道第一条消息也是正确的。

Bob 知道 DH 参数是安全的。别忘了，这是 Bob 自己选取的。因此，正如 Alice 一样，他确信只有知道 y 满足 $g^y = Y$ 的人才能够计算出最终的密钥 k。Alice 认证了她发送的 Y，因此她是惟一知道相应的 y 的人。这使 Bob 确信 Alice 是另外一个能计算出密钥 k 的人。

15.8.3 攻击者的观点

最后，我们从攻击者的观点看一下这个协议。如果对通信进行监听，可以看到 Alice 和 Bob 交换的所有消息。但是，密钥 k 是使用 DH 协议计算出来的，因此只要 DH 参数是安全的，像这样的被动攻击不能揭示出任何有关密钥 k 的信息。换言之，我们必须尝试做一个主动攻击者。

一个有意义的操作是查看每一个数据元素并试图改变它们。此时，我们很快就会因为两个认证证据而停下来。Alice 的最终认证涉及 Alice 和 Bob 之间交换的所有数据，这意味着我们除

了尝试重放攻击来重放以前记录下来的协议执行之外不能改变任何数据元素。但是随机的nonce和随机的X值阻止了任何重复攻击尝试。

这并不意味着我们就不能进行任何尝试了。比如，我们可以把s_a赋予一个较大的值。只要Bob能够接受这个大的值，大多数协议会正常完成。这里有3个问题：首先，增大s_a并不是攻击，因为它仅仅使DH素数变得更大，从而使DH参数更加强健。第二个和第三个问题是两个认证将会失败。

在现实世界中，你会发现很多协议中有未经认证的数据元素。多数设计者不会对协议中的s_a进行认证，因为改变s_a并不会导致攻击（Alice和Bob各自检查p的长度足够大就可以了）。允许攻击者进行尝试总不是个好主意。除了必需的，我们不愿给他们任何更多的工具。我们当然可以想像没有认证s_a将是危险的。例如，假设Bob喜欢使用程序内置列表中的DH参数，并且只是在需要的时候才产生新的参数。只要Alice和Bob选取的DH素数的长度仍在列表中，Bob就不会生成新的参数集。但这也意味着，Bob的参数生成代码和Alice的参数验证代码从没有被使用过，因此不大可能被正确地测试。参数生成和测试代码中的bug仍然处于隐藏状态，直到攻击者增加了s_a的值。的确，这是一个不大可能出现的情形，但就是存在成千上万的不大可能的情形，它们对安全来说都是有害的。数以千计的低可能性的风险加起来就成为高概率的风险。这就是我们要如此偏执地在任何时候都要阻止我们能够阻止的任何类型攻击的原因。

15.8.4　密钥的泄露

那么，如果系统的另外某些部分的安全性遭到了威胁会发生什么呢？

如果Alice只是丢失了她的认证密钥，而攻击者却不知情，Alice所丢失的只是运行这一协议的能力，她仍可以使用已经建立起来的会话密钥。这正是你所期望的协议行为。如果Bob丢失了密钥，情况是一样的。

如果Alice丢失了会话密钥，而攻击者并不知道，她必须和Bob一起再次运行密钥协商协议来建立一个新的会话密钥。

如果攻击者设法获知了密钥，事情会变得很糟糕。如果Alice的认证密钥泄露了，从那一刻起攻击者就可以冒充Alice，直到Bob被告知并停止接受Alice的认证。这种结果是不可避免的。如果你丢失了汽车钥匙，任何找到钥匙的人都可以使用你的汽车。这就是钥匙的主要作用之一：允许使用某些功能。这个协议的确有这样理想的性质：Alice和Bob之间过去的通信仍然是保密的。即使知道了Alice的认证密钥，并不会使攻击者找到已经完成的协议的会话密钥，即便攻击者记录下了所有的消息。这个性质称为前向保密性[①]。关于Bob的认证密钥，同样的性质也是成立的。

① 有时会看到完善的前向保密性这个术语，或者PFS，但我们不使用类似“完善”之类的词，因为它们绝不是完善的。

最后，我们考虑会话密钥泄露的情形。密钥 k 是 g^{xy} 的 Hash 值，其中 x 和 y 都是随机选取的。这不会提供其他任何密钥的信息，当然它也不能提供关于 Alice 和 Bob 的认证密钥的任何信息。一个协议运行中的 k 值与其他协议运行中的 k 值是完全无关的（至少，当我们假定 Alice 和 Bob 使用一个好的 PRNG 时是这样的）。

关于密钥泄露，我们的协议提供了可能的最好保护。

15.9 协议的计算复杂性

下面看一看我们的解决方案的计算复杂性。假设DH参数的选择和验证都进行了缓存，所以在计算单个协议运行的工作量时不考虑它们。剩下来的就是下面的计算：

- Alice 和 Bob 都需要 3 次 DH 子群中的指数运算。
- 一次认证生成。
- 一次认证验证。
- 各种非常高效的操作，比如随机数生成、比较操作、Hash 函数。

如果使用对称密钥的认证，那么协议的运行时间主要由DH指数操作控制。让我们看一看这需要多少工作量。Bob 和 Alice 每个人需要做 3 次指数为 256 比特的模指数运算，这大约需要 1150 次模乘法运算[①]。要了解这到底需要多少工作量，我们把它与 RSA 签名的计算代价进行比较，其中RSA模与DH素数的长度是一样的。对于一个 s 比特的模，如果不使用CRT（中国剩余定理），签名算法需要 $3s/2$ 次乘法运算。使用 CRT 会节省 1/4 的时间，所以对一个 s 比特的数计算 RSA 签名的代价类似于做 $3s/8$ 次乘法运算的代价。这带给我们一个有趣的结论：当模比较大时，计算RSA签名与DH计算相比要相对慢一些，而当模较小时会相对快一些。二者的平衡点大约是 3000 比特。这是因为 DH 总是使用 256 比特的指数，而 RSA 的指数则随着模的大小而增大。

由此得出结论，对于我们使用的公钥长度，DH的计算代价与RSA签名的计算代价基本上是一样的。这仍然是协议计算中的主要因素，然而却是相当合理的。

如果RSA签名用于认证，计算负荷的工作量差不多要加倍。（我们可以忽略RSA的验证，因为它非常快速。）这仍然不是过分的。CPU 的速度在飞速增长，在大多数实际实现中你会看到，通信延迟和开销比计算占了更多的时间。

15.9.1 优化技巧

有一些优化可以用于DH操作。受到加法链的启发，每一次指数运算可以使用更少的乘法运算。此外，Alice 既要计算 X^q 又要计算 X^y。使用加法序列的方法同时计算这两个结果可以节省大约 250 次乘法运算。详细的讨论请参见[10，ch.4]。

① 对于简单的二元指数运算来说是这样的。更好的优化算法可以把它降低到 1000 次乘法运算。

另外还有各种技巧可以使生成y和计算g^y的速度更快，但是这些技巧需要很多额外的系统复杂性，因此我们不愿使用它们。

15.10　协议的复杂性

这个协议也是一个极好的例子，说明了为什么说协议设计是及其困难的。即使像这样简单的协议也很快会充满一页纸，并且我们还没有包括用于DH参数生成或者认证方案检查的所有规则，这些规则从抽象的层次来说还是不确知的。然而，要明了所有要做的事情已经很困难了，而且更复杂的协议会变得更加庞大。Niels工作过的一个特殊的智能卡支付系统大约有12个协议，使用了50页的符号和协议规范进行详细描述，并且是使用专用的高度简化了的记号！另外还用了密密写满文字的50页文档来描述鉴定的安全性实现问题。

一组密码协议的完整文件会达到几百页。很快，协议就会变得太复杂了，以至于不能用脑子记住它，这是危险的。一旦你不能完全理解它，几乎不可避免地会出现弱点。上面所提到的计划可能过于复杂以至于不能被完全理解，即使是设计者本人。

几年过后，Niels从事另一个商业性智能卡系统的工作。这是一个著名的已经建立起来的系统，广泛应用于许多不同的智能卡应用程序中。一天，同事Marius Schilder提出了一个问题——更正确地说，是系统中的一个大漏洞。结果表明，其中的两个协议产生了破坏性的相互冲突。其中的一个协议根据长期的卡密钥计算会话密钥，这有点儿像本章所描述的密钥协商协议。另一个协议则根据长期的卡密钥计算认证值。稍做调整，就可以使用第二个协议来让智能卡计算会话密钥，并把其中的一半比特发送给你。知道了一半的密钥比特，攻击系统的其余部分就微不足道了。虽然这个漏洞在第二个版本中得到了修补，但是它很好地说明了大协议规范中的问题。

现实世界中的系统总是有很大的协议规范。通信是非常复杂的，而增加密码函数和不信任使得事情变得更加困难。我们的建议是：一定要非常小心地对待协议的复杂性。

这个领域中的一个基本问题是，对于协议来说没有好的模块化的标记，所以每件事情最后都混在一起。在本章中我们已经看到了这一点：DH参数协商、DH密钥交换以及认证都结合在一起。这不是各个松散部件的简单组合，而是要把规范和实现调整在一起。它看起来更像是一个真正有害的复杂的计算机程序，而没有经过任何模块化。我们都知道那将导致什么后果，但我们已经开发出模块化的技巧来处理程序的复杂性。遗憾的是，对于协议我们还缺乏模块化的技术。假如你正在寻找一个长期的研究计划，这很可能就是一个。另一方面，Niels曾经针对这个问题写过一个研究提议，并且被接受了。他得到了4年的研究资助，但他最终没有承担这个课题，因为他意识到甚至没有办法开始这个课题的研究。得到这个工作的人也没有在这一课题上有所进展，而是在另一个领域做了4年有价值的研究。这个教训表明，它不是一个容易的课题。如果经过数年的研究之后得到的结论是："我没有任何线索"，那么将很难拿到那个博士学位了。

15.11 善意的警告

我们尽力使协议的设计看起来尽可能容易。请不要被这一点欺骗，协议的设计是及其困难的，而且需要经过大量的体验。即使有很多的经验，也很容易出错。尽管我们在本书中非常努力地使事情回复正常，但我们这里所设计的密钥协商协议也存在出错的可能性。

15.12 使用口令的密钥协商

到目前为止，我们假定存在作为密钥协商基础的认证系统。在许多情况下，你所拥有的只是一个口令。可以使用口令作为MAC的密钥来运行这个协议，但是存在一个问题：得到了这个协议的记录（通过对通信进行窃听而得到），就可以测试任何特殊的口令，只需要计算认证值并看看它是否正确就可以了。

使用口令带来的问题在于，人们并不是从一个大集合中选取口令。有一些程序可以搜遍所有可能的口令。对于一个我们希望的理想的密钥协商协议来说，窃听者不能执行离线的字典攻击。

这样的协议是存在的，最有名的例子大概是SRP [97]。他们提出了一个重要的安全性改进方案。遗憾的是，整个领域都是专利的雷区。我们还没有发现一个合适的没有与专利捆绑在一起的基于口令的密钥协商协议。斯坦福正在放弃关于SRP的受限使用的专利许可。这是一个不透明的复杂问题，我们最好离它远一点儿。如果我们对实现这些协议都不感觉安心，那么就不会把读者引入这个诱惑。

第16章 实现问题（II）

我们设计的密钥协商协议带来了一些新的实现问题。

16.1 大整数算术

公钥计算都依赖于大整数算术。正如我们以前提到的，正确地实现大整数算术并不是一件易事。

大整数例程几乎总是与平台有关的。通过使用特定平台的性质而得到的效率都很大，因而不能忽视。例如，大多数CPU使用带进位加来实现多字（multiword）值的加法。然而，在C或者几乎所有其他高级编程语言中不能访问这一指令。在高级语言中做大整数算术通常比平台上的优化实现要慢几倍，而且，这些计算也形成了公钥性能上的瓶颈，所以对大整数例程来说，使用指定平台的代码将带来很大的收益。

我们不打算详细介绍如何实现大整数算术。其他一些书（比如Knuth [55]）已经有很好的介绍。对我们来说，真正的问题在于如何测试大整数算术。

在密码学中，大多数实现者有着不同的目的。我们认为2^{-64}（大约为1/1800万兆）的出错率是不可接受的，而能够达到这样的出错率时大多数工程师都是很高兴的。很多编程人员似乎认为2^{-20}的出错率是可以接受的，甚至认为是很好的。然而我们必须做得更好，因为我们是在有敌手的环境下工作。

多数分组密码和Hash函数都很容易测试，很少有bug会导致难以发现的错误。如果在AES的S-box表中犯了一个错误，通过几次AES加密测试就可以检测到。简单的随机测试就使用了分组密码或者Hash函数中的所有数据通路，并且很快发现所有的系统问题。所使用的代码通路不依赖于提供的数据，或者只限于有限的方式。对称模块的任何测试将使用所有可能的执行控制流。

大整数算术是不同的。主要的差别在于：多数实现中，代码通路依赖于数据。很少使用到传导最后进位的代码。除法例程常常使用一块代码，而2^{32}次除法甚至2^{64}次除法中才使用一次该代码。随机测试将不会发现这部分代码中的bug。当我们使用大CPU时，这个问题变得更糟糕。在32比特的CPU上，还可以运行2^{40}次随机测试，期望在数据通路的每一部分都会有一个32比特的word值出现。但是这一类型的测试对64比特的CPU就不起作用了。

作为推论，你必须极其小心地做大整数算术例程的测试。你必须验证在测试期间每一个代码通路实际上都取到了。要做到这一点，必须精心设计测试向量。不仅需要使用每个代码通路，

也要遍历所有的边界条件。如果要测试 $a < b$，那么应该对 $a = b - 1$、$a = b$ 和 $a = b + 1$ 进行测试（当然，仅当这些条件可能满足时）。

最优化使得这个已经很坏的情况更加糟糕。由于这些例程是性能瓶颈的一部分，该代码往往被高度优化。而这又会导致更多的特殊情况，更多的代码通路，等等，所有这些都使测试工作变得更加困难。

一个简单的算法错误可以产生灾难性的安全后果。下面是一个例子：当Alice正在计算RSA签名时，在模 p 而不是模 q 求幂时有一个小错误。（她正在使用CRT来加速其签名。）她发送出去的是 $\sigma + kq$，而不是正确的签名 σ，其中 k 是某一个值。（Alice得到的模 q 结果是正确的，但模 p 是错误的，所以它一定是 $\sigma + kq$ 的形式。）攻击者知道 $\sigma^3 \bmod n$，这是Alice要计算其根的那个数，它只依赖于消息。但是，$(\sigma + kq)^3 - \sigma^3$ 是 q 的倍数，把这个数与 n 取最大公因子就暴露了 q，因而暴露了 n 的分解。这是一个灾难！

那么我们要怎么做呢？首先，不要实现你自己的大整数例程，而应使用已有的库。如果你想在这上面花时间，就花些时间理解和测试已经存在的库。第二，在你的库中运行真正好的测试。要确保测试了每一个可能的代码通路。第三，在应用中加入额外的测试。有几个技巧可以使用。

16.1.1 wooping

这一节描述的技巧有着与众不同的名字：wooping。David Chaum和Jurjen Bos在一次热烈的讨论中，突然发现需要给一个特殊的验证值命名。在情急之下，他们中的一个人建议取名为"woop"，之后这个名字就代表了整个技巧。Bos随后在他的博士论文[10，ch.6]中详细描述了这一技巧，但是因为它不够学术而舍弃了这个名字。

wooping的基本思想是检验随机选取的小素数的模计算，把它作为一个密码问题来考虑。我们有一个大整数库，它设法进行欺骗，给我们以错误的结果。我们的任务是检查是否得到了正确的结果。只在同一个库中检查其结果可不是个好主意，因为该库可能会产生一致性的错误。使用wooping技巧，我们就可以检验库中的计算，只要我们假定它不是常有这样的恶意，即费尽心机试图破坏我们的验证计算。

首先，我们产生一个相对较小的随机素数 t，其长度为64~128比特。t 值不应该是固定的或者是可预测的，不过这正是PRNG要做的。这个 t 值要对所有的参与方保密。那么，对于在计算中出现的任意的大整数 x，我们保存 $\tilde{x} := (x \bmod t)$。值 $\tilde{x}$ 称为 x 的woop。这个woop值是固定长度的，一般比大整数要小很多，因此计算woop值的代价不是特别大。

于是我们需要保存每个整数的woop值。对算法的每一个输入 x，我们直接计算 $\tilde{x}$ 为 $x \bmod t$。对于所有的内部计算，我们把大整数计算镜像为woop值来计算woop结果，而不是通过大整数结果来计算它。

正常的加法计算为 $c := a + b$。我们可以通过 $\tilde{c} = \tilde{a} + \tilde{b}$ 来计算 $\tilde{c} \pmod t$。乘法可以用同样的方法处理。在每一次加法或者乘法计算之后，可以通过检查 $c \bmod t = \tilde{c}$ 来检验 $\tilde{c}$ 的正确性，然而，到最后再做所有的检查将更加高效。

模加法就稍微困难一点。我们不写 $c = (a+b) \bmod n$，而是记为 $c = a + b + k \cdot n$，其中 k 的选取要使得 c 位于 $0, \cdots, n-1$ 之内。这只是用另一种方式来表示模约化。假定 a 和 b 都在 $0, \cdots, n-1$ 之内，那么在这种情况下 k 要么等于 0 要么等于 -1。其 woop 形式为 $\tilde{c} = (\tilde{a} + \tilde{b} + \tilde{k} \cdot n) \bmod t$。在模加法程序内的某个地方，$k$ 值是已知的，我们所要做的是说服该库向我们提供 k 值，以便计算 $\tilde{k}$。

模乘法做起来就有些困难了。我们还需要书写 $c = a \cdot b + k \cdot n$，为了计算 $\tilde{c} = \tilde{a} \cdot \tilde{b} + \tilde{k} \cdot \tilde{n} \pmod t$，需要 $\tilde{a}$、$\tilde{b}$、$\tilde{n}$ 和 $\tilde{k}$。前 3 个值马上就可以得到，但是 $\tilde{k}$ 必须以某种方式从模乘法程序中弄到。在创建该库时可以做到这一点，但是很难对一个库进行改型。一般的方法是先计算 $a \cdot b$，然后使用长除法将它除以 n。这个除法的商就是 woop 计算所需要的 k。余数是结果 c，这个一般方法的缺点是太慢。

一旦可以对模乘法保存woop值，对于模指数也就容易做了。模指数程序只不过是通过模乘法来构造模指数。（有人使用单独的模平方程序，但是那也可以扩展到 woop 值，就像模乘法程序一样。）对每个整数都保存 woop 值，让每一次乘法都根据输入的 woop 计算其结果的 woop。

扩展 woop 算法是基于输入的 woop 值来计算结果的 woop 值：如果有一个或者多个 woop 输入是错误的，那么 woop 输出几乎一定也是错误的。所以，一旦一个 woop 值不正确，这个错误将传播到最终结果。

我们在计算结束时检查 woop 值。如果结果是 x，需要做的就是检查$(x \bmod t) = \tilde{x}$。如果这个库发生了错误，这个 woop 值将不会匹配。我们假定该库不会以依赖于我们选择 t 的方式精心处理其错误。毕竟，在我们选择 t 之前库代码已经确定很久了，而且库代码不受攻击者的控制。容易说明的是，库中可能存在的任何错误将被绝大多数 t 值捕获。所以，对现有的库增加 woop 验证将使我们能很好地核实这些计算。

我们真正想要的是具有内置 woop 验证系统的大整数库，但我们不知道是否有这样的库。

你的woop值应该是多大呢？这取决于多种因素。对于随机错误，woop值检测不到错误的概率大约是 $1/t$。但在我们的世界里没有任何事情是完全随机的。假设库中有一个软件错误，而且假定攻击者知道这个错误，他可以选择计算的输入，从而不仅引发这个错误，而且也可以选择错误所导致的差别。这就是 t 为什么必须是随机的、保密的数。不知道这个 t，攻击者不可能锁定最终结果中的错误并选择一个差值而不被我们的 wooping 察觉到。

那么，如果你是个攻击者，你该怎么做呢？你当然要设法引发错误，但是你也要试图迫使其差值模 t 等于零，而且对尽可能多的 t 值取模。最简单的对策是要求 t 是素数。如果攻击者想要欺骗 16 个不同的 64 比特的素数，那么他至少需要精心选择 $16 \cdot 64 = 1024$ 比特的输入。在多数计算中，由于攻击者能够选择的输入比特个数是有限的，这就限制了攻击成功的可能性。

t 值越大越好。存在如此之多的大素数，使得攻击者成功的可能性很快就消失了。如果要保持我们初始的 128 比特的安全目标，就需要 128 比特的 t，或者在此范围内的值。

woop值不是系统首要的安全性，它们只是一种支持。如果一个woop验证失败了，我们知道软件中有一个bug需要修复。程序应该终止它所做的事情并报告一个重大错误。这也使得攻击者更加难以对系统发动重放攻击。因此，我们建议使用64比特的素数作为t值。同128比特的素数相比，这大大地减少了开销，而且在实践中也足够好了。如果承担不了64比特的woop，那么32比特的woop总是比什么也没有要好。特别是在多数32比特的CPU上，由于可以直接使用乘法和除法指令，从而可以非常有效地计算32比特的woop。

如果你曾经就攻击者会提供的大量的数据做过计算的话，那么你也应该检查中间的woop值。每一个检查非常简单：$(x \bmod t) \stackrel{?}{=} \tilde{x}$。中间值仅仅依赖于攻击者提供的有限比特，通过检查中间值，可以使攻击者更加难以欺骗woop系统。

我们非常喜欢使用带有woop验证的大整数库。这是一个相对简单的方法，而且避免了大量潜在的安全问题。同时我们相信，对数据库增加一次woop验证的工作量比对于使用数据库的每一个应用程序增加特定应用的验证工作量要小得多。

16.1.2 检测DH的计算

如果你的库不支持woop，你将不得不放弃使用它。我们已经描述的DH协议包含大量的检查，即计算结果不能为1，同时计算结果的阶应该是q。遗憾的是，这些检查不是由进行计算的一方执行的，而是由接收该计算结果的一方执行的。总体来说，你不希望发送任何不正确的结果，因为它们可能泄露信息，但是在这一特殊情况下，它看起来并没有太大的危害。如果结果是错误的话，协议会以某种形式失败，于是就会注意到这个错误。在要求计算g^x的值时，如果算术库返回x，协议的安全性这时才遭到破坏。这是一种典型的错误，采用通常的测试就很容易检测到。

如果需要的话，我们可以在没有woop检测的库中运行DH协议。这里我们担心这种罕见的算术错误不大可能在进行g^x的计算时泄露x的值。其他的错误似乎没有什么危害，尤其是因为DH计算不涉及长期的秘密。不过，如果可能，我们仍然会使用woop库，也许这样会感到安全一些。

16.1.3 RSA加密的检测

RSA加密更容易受到攻击，因此需要额外的检查。如果发生错误，可能会泄露你所加密的信息，甚至泄露你的私钥。

如果不能使用woop验证，这里有另外的两种方法可检查RSA加密。假设实际的RSA加密包括计算$c = m^5 \bmod n$，其中m是消息，c是密文。为了验证这一点，我们可以计算$c^{1/5} \bmod n$并且把它和m进行比较。其缺点是，相对于快速计算来说，这是一个非常缓慢的验证过程，同时它要求具有私钥的知识，这恰好是我们在进行RSA加密时所不具备的。

或许一个更好的方法是选取一个随机数z，并且检查$c \cdot z^5 = (m \cdot z)^5 \bmod n$。这里我们有3个五次幂要计算：$c = m^5$、计算$z^5$以及最后比较$(mz)^5$和$c \cdot z^5$是否相等。随机的算术错误很容

易被这一验证检测到。通过选择随机的 z，使得任何攻击者不可能拿一个错误产生的值来进行攻击。在我们的设计中，我们仅仅使用 RSA 加密算法来加密一个随机值，因此攻击者根本不可能把它作为一个攻击目标。

16.1.4　检查 RSA 签名

RSA签名很容易检查，签名者只需要运行签名验证算法就可以了。这是一个相当快速的验证方案，同时，算数上的错误很容易被察觉。每一次计算 RSA 签名的时候都应该检查刚刚产生的签名，我们没有理由不这样做。

16.1.5　结论

我们讨论的检查是对于大整数库的正常测试之外的检查，它并不能代替正常的检测，任何软件（尤其是安全软件）都必须进行检查。

如果这些检查中的任何一个曾经失败，就说明你的软件不合格。在这种情况下，你就不能做什么了。继续下来的工作是不安全的，而你一点儿也不知道出现了什么类型的软件错误。你真正能做的只有记录错误，同时终止程序。

16.2　更快的乘法

这里有很多方法可以进行模乘法运算，它们比进行一次完整的乘法然后再进行长除法的方法更快。如果必须进行大量的乘法计算，那么 Montgomery [68]的方法是最为广泛应用的一个算法。遗憾的是，Montgomery 的文章是很难懂的，读者可以参见[26]，该文献对于他的方法给出了更易读的描述。

Montgomery 方法的基本思想是计算（$x \bmod n$）的一个技巧，其中 x 比 n 大很多。传统的“长除法”方法是从 x 里面减去一个 n 的合适倍数。Montgomery 的思想更简单：用 x 重复除以 2。如果 x 是偶数，通过把它的二元表示向右移一个比特来实现 x 除以 2。如果 x 是奇数，我们首先加 n（当然，并不改变模 n 的值），然后把得到的偶数结果除以 2（这种方法仅仅在 n 是奇数时有效，而在我们的系统中 n 总是奇数。对于偶数的 n 值可以做简单的推广）。如果 n 是 k 比特长，且 x 不超过$(n-1)^2$，我们总共执行 k 次除以 2 的除法。结果总是在区间 $0, \cdots, 2n-1$ 中，这几乎是完全约化了的模 n 的结果。

不过要注意，我们是在除以2，这将得到一个错误的答案。Montgomery 的约化没有真正得到$(x \bmod n)$，而是 $x/2^k \bmod n$，其中 k 是一个合适的值。这个约化更快速，但是得到了额外的因子 2^{-k}。有许多种技巧可以处理这个额外的因子。

简单重新定义协议是一个不好的做法，这种方法在计算中包括额外因子 2^{-k}。这样做不好，因为它混淆了不同的层次，修改了密码协议规范来支持一个特殊的实现技巧。也许以后你想在另外一个平台上实现这个协议，但是发现根本不想使用 Montgomery 乘法（也许那个平台比较

慢，但是有一个大整数处理器直接执行模乘法运算）。在这种情况下，协议中的2^{-k}因子就成了一个真正的障碍。

通常使用的技巧是改变数的表示方法。数x的内部表示为$x \cdot 2^k$。如果想要把x和y相乘，就对它们各自的表示做Montgomery乘法。得到的是$x \cdot 2^k \cdot y \cdot 2^k$，但同时从Montgomery约化中得到了额外的$2^{-k}$因子，于是最终的结果是$x \cdot y \cdot 2^k \bmod n$，这正好是$xy$的表示。使用Montgomery约化的开销代价包括：把输入的数值转换成内部表示（乘以2^k）的代价，以及把输出转换回实际结果（除以2^k）的代价。第一个转换可以通过执行x和（$2^{2k} \bmod n$）的Montgomery乘法完成，第二个转换只需要对另外的k比特进行Montgomery约化，因而它被2^k除。Montgomery约化的最终结果不能保证比n小，但在大多数情况下它比$2n-1$小。在此情况下，简单的检验和一次可选的减去n的减法就得到了最终的正确结果。

在实际实现中，Montgomery约化从来没有按比特进行的。假设CPU使用w比特的字。给定值x，找到一个小整数z满足$x + zn$的最低位字全部是零。可以证明z就是一个字，可以通过把x的最低位字与一个只依赖于n的单字因子相乘计算得到。一旦$x + zn$的最低位字是零，就除以2^w，这通过把该值向右移动一个字来完成，因此比按比特的实现要快很多。

16.3 侧信道攻击

我们在9.5节简单地讨论了计时攻击和其他的侧信道攻击。在那里只进行简单讨论的主要原因不是因为这些攻击是良性的，相反，计时攻击对于公钥计算来说是最有用的。

正如已经提到的那样，对称密钥的计算对每一次计算往往具有简单的代码路径，计算分组密码所花费的时间是一个常量。当然，是相对的常量。各种各样的与缓存有关的可变计时几乎是不可能预测的。我们还没有看到有人在类似PC的设备上使用缓存导出的计时差异来攻击对称密码系统，尽管这在理论上是可行的。

有的密码引起的实现使用不同的代码路径来处理特殊情形。IDEA[61, 62]和MARS[13]就是两个例子。其他密码使用CPU操作，它的时序根据所处理的数据而变化。在有的CPU上，乘法（RC6[78]和MARS使用）或者数据轮换（在RC6和RC5[80]中使用）的执行时间依赖于输入数据，这可能导致计时攻击。本书介绍的模块不使用这些操作。

公钥密码系统更容易遭受计时攻击。公钥操作的代码路径常常与数据有关。对于不同的数据，总是会产生不同的处理时间，而这些计时信息又会导致攻击。设想一下电子商务中的安全Web服务器。作为SSL协商的一部分，服务器需要解密客户端选择的RSA消息，因此攻击者可以连接到服务器，要求它解密所选择的RSA值，然后等待响应。服务器响应所需要的确切时间就给了攻击者一些重要的信息。实际上，如果密钥的某个比特是1，来自集合A的输入常常比来自集合B的输入的响应要快一些，而如果该密钥的比特是0，则没有差异。攻击者就可以利用这个差异来攻击系统。他从集合A和集合B中都产生数百万个询问，试图从这两个群的

响应时间上找到一种统计差异。可能有其他的因素影响到确切的响应时间，但是可以通过足够多的询问对它们进行平均。最终，你收集了足够的数据来测量对A和B的响应时间是否不同。这样，攻击者就得到了1比特的密钥信息，然后攻击者设法获得下一个比特。

这些听起来很牵强，但是在实验室里已经实现了，而且在现实中也很可能做到[56]。

16.3.1 对策

有几种方法可以保护自己免受计时攻击。最明显的办法是保证每一次计算都要有一个固定的时间，这要求整个库按照这一目的进行设计。此外，有一些资源的计时差异几乎是不可能控制的。一些CPU的乘法指令对于一些数的运算速度总是比对其他数的快一些。多数CPU都有复杂的缓存系统，所以，只要内存访问方式依赖于不公开的数据，缓存延迟就可能带来计时差异。要消除公钥操作的所有计时差异几乎是不可能的，因此我们需要其他的解决方案。

我们很容易想到在每一次计算的最后增加随机的延迟，但是这并没有消除计时差异，它只是把这种差异隐藏在延迟的噪音中。攻击者可以采用更多的例子（比如利用你的机器做更多的计算），从而平均这些结果并期望平衡掉增加的随机延迟。攻击者需要的确切尝试次数依赖于攻击者寻求的计时差异的量级，以及增加的随机延迟的量级。在实际的计时攻击中，几乎总是有大量的噪音，所有任何试图进行计时攻击的攻击者已经进行了平衡。惟一的问题在于信号与噪声的比率。

第三种方法是把操作常数化，强迫它持续一段标准的时间。在开发中间，选择一个持续时间d，它比计算所需要的时间更长。然后，在计算开始的时候标记时间t，计算完成之后还需一直等待到时间$t+d$。这有一点儿浪费，但是还不坏。我们喜欢这个解决方案，但是它只对纯粹的计时攻击提供保护。如果攻击者可以监听你的机器发出的RF辐射，或者测量到能量消耗，那么计算与延迟之间的差异大概是可以探测到的，从而允许计时攻击以及其他的攻击。当然，基于RF的攻击要求攻击者能实际地接近机器。同可以通过因特网发动的计时攻击相比，这极大地降低了威胁。

此外还可以使用由盲签名[56]导出的技巧。对于有些类型的计算它们（几乎）可以隐藏所有的计时变化。

对于计时攻击问题没有完善的解决办法。很明显，保护你所购买的计算机免受复杂的攻击是不大可能的，比如基于RF的攻击。但是，即使不能创造一个完善的解决方案，也可以得到一个相当合理的方案。只不过要特别小心公钥操作的计时。与仅仅使公钥操作有一个固定的时间相比，一个更好的办法是利用上面提到的技巧使得整个处理过程有一个固定的时间。也就是说，不仅使公钥操作有固定的时间，而且把请求进入到响应输出之间的时间也固定下来。如果请求在时间t进入，那么就在时间$t+C$发出响应，其中C是一个常数。不过，要确保绝不泄露任何计时信息，最好保证响应在时间$t+C$已经就绪。为了保证这一点，大概需要限制你接受进入请求的频率为某一个固定的上界。

16.4 协议

实现密码协议与实现通信协议没有多大差别。最简单的办法是在程序计数器中保持状态，然后按照顺序依次执行协议的每一步。除非使用多线程，否则当等待回应的时候，则停止程序中所有其他事情。由于回应可能不会马上到来，这通常不是一个好主意。

更好的办法是保持一个确切的协议状态，每当消息到达的时候就升级协议状态。这个基于消息驱动的方法在实现时需要稍微多做一些工作，但是它提供了更高的灵活性。

16.4.1 安全信道上的协议

大多数密码协议在不安全的信道上执行，但有时也会在安全的信道上运行密码协议。这在某些情形中是有意义的。比如，每一个用户与密钥分配中心之间有一个安全信道，密钥分配中心使用一个简单的协议来为用户分配密钥，从而允许他们与其他人进行通信（Kerberos协议所做的事情与此有些类似）。如果你与一个参与方执行密码协议，而你已经与他交换了密钥，那么就应该使用完全的安全信道功能。特别是你应该实现重放保护。这很容易做到，并且它避免了大量的对于密码协议的攻击。

有时安全信道允许协议使用捷径。例如，在安全信道提供了重放保护而协议本身没有提供时。当然，旧的模块化的规则认为，协议应该使它对于安全信道的依赖性达到最小化。

在以后的协议实现讨论中，我们假定协议是在不安全的信道上执行的。其中有些讨论不适用于安全信道的情况，但是该解决方案绝不会有危害。

16.4.2 接收消息

当协议状态显示接收到一个消息时，有几个检查必须要做。第一个是查看该消息是否属于这个协议。每一个消息都应该以下面的字段开始：

协议标识符：明确标识这是哪一个协议以及哪一个版本。版本标识是很重要的。

协议实例标识符：标识该消息属于协议的哪一个实例。也许Alice和Bob正在同时运行两个密钥协商协议，我们不希望混淆这两个运行。

消息标识符：标识协议中的消息。最简单的办法就是对它们进行编号。

根据情况的不同，某些标识符可能是隐式的。比如，对于在自己的TCP链接上运行的协议，端口数和相关的套接字（socket）惟一地标识了协议的实例。协议标识符和版本信息只需要交换一次。注意，至少交换一次很重要，这样可以保证它们包含在协议的任何认证或者签名之中。

检查完协议标识和实例标识之后，我们知道了发送消息的协议状态。让我们假定协议状态刚刚发送了消息 $n-1$，正在等待接收消息 n。

如果接收的消息的确是消息n，事情就很容易了。只需要按照协议规则指定的方法来处理即可。但是，如果它是另外一个数，该怎么办呢？

如果这个数比n大或者比n–1小，就会发生非常奇怪的事情。这样的消息不应该产生，因此一定是某种伪造消息。你一定不要理睬这个伪造消息的内容。

如果接收的消息是n–1，那么你发送的消息n可能没有到达。至少，如果你在一个不可靠的传输系统上运行协议，会发生这种情况。因为我们希望使对于系统其他部分的依赖性最小化，这正好满足我们的假定。

首先，检查新接收的消息n–1绝对与先前接收的编号为n–1的消息是一样的。如果它们不一样，必须忽略这个新消息。发送第二个应答将破坏很多协议的安全性。如果消息是一样的，重新发送消息n。当然，重新发送的版本必须和以前发送消息n的版本是一样的。

如果由于这些规则中的任何一条使你忽略了接收到的消息，你要做第二个决定。你应该终止协议吗？答案在某种程度上取决于应用和具体情况。如果在安全信道上运行协议，一定是出错了。要么安全信道的安全性受到了危害，要么你进行谈话的参与方在捣鬼。无论哪一种情形，都应该终止协议和信道。完全删除协议状态和信道状态，包括信道密钥。

如果你是在不安全的信道上执行协议，那么任何忽略的消息都可能来自于攻击者，他试图干涉协议。理想情况下，应该忽略攻击者的消息并且完成协议。当然，这并不总是可以做到的。比如，如果攻击者伪造的消息n–1先到达你这里，你会发送应答。如果以后收到了“真正”的消息n–1，就强迫你忽略它。在这个情形中无法恢复，因为你不能安全地发送第二个应答。但是，你一点儿也不知道接收到的两个n–1消息中哪一个是真的，所以，为了具有最大的机会成功地完成协议，你应该把第二个n–1记录为错误，并与往常一样继续协议。如果你回复的消息来自于攻击者，协议最终会失败，因为密码协议的设计尤其要防止攻击者成功地与一个参与方完成协议。

16.4.3 超时设定

任何协议都包括超时设定。如果在合理的时间内没有得到消息的响应，可以重新发送最后一条消息。几次重新发送之后，你必须放弃。当你不能与另外一方进行通信时，就没有办法继续执行协议了。

实现超时设定的最简单的方法是发送计时消息给协议状态。可以使用协议明确设置的计时器，或者使用大约几秒前发送的计时消息。

一个著名的攻击是发送“start-of-protocol”这一消息给特定的机器。每一次收到start-of-protocol消息时，你初始化一个新的协议执行状态。在接收到几百万条这样的消息之后，你就用尽了内存，此时所有事情都停止了。一个好的例子是SYN溢出攻击。目前还没有什么容易的方法可以保护自己不受这种溢出攻击，但是它们的确表明了删除旧的协议状态的重要性。如果协议的延迟时间太久，我们就应该删除它。

用于重新发送的合理计时是可以讨论的。根据我们的经验，因特网上的一个包要么在几秒钟左右到达，要么就永远丢失了。如果在5秒钟之内没有收到答复，重新发送一个消息是合理的。三次重试应该是足够了。如果消息丢失率如此之高，相继丢失4条消息的时间超过了15秒，那么你一定不愿意再在这次连接上丢失所有的数据包了。我们倾向于在20秒之后向用户通知这一问题，而不是要求用户坐在那里等上一两分钟。

3

第三部分

密 钥 管 理

第 17 章　时钟

第 18 章　密钥服务器

第 19 章　PKI 之梦

第 20 章　PKI 的现实问题

第 21 章　PKI 的实用性

第 22 章　存储秘密

第17章 时　　钟

在我们开始详细讨论密钥管理之前，需要讨论另一个功能：时钟。乍看起来，这绝对不是密码方面的问题，但因为当前时间常常被用在密码系统中，所以我们需要一个可靠的时钟。

17.1 使用时钟

对于时钟，有几种密码方面的使用。密钥管理函数常常与最终使用期限相关联。当前时间能够提供一个惟一值和完整的事件发生顺序。我们将详细地讨论每一种使用。

17.1.1 有效期

在很多情况下，我们要限制文件的有效期。在现实生活中，我们也会看到被限制的有效期。支票、飞机票、凭证、优惠券甚至是版权都有有效期。限制数字文件有效期的标准方式是在文件本身包含有效期。但为了检查一个文件是否到期，我们需要知道当前时间，因此就要用到时钟。

17.1.2 惟一值

时钟另一个有用的功能是可提供一个机器的惟一值。我们已经在几个地方用到了瞬时值。瞬时值的主要特点是任何一个值不能使用两次，至少在定义的范围内是这样的。有时范围被限定了，例如我们在安全信道中使用的瞬时值，这时它就能用计数器来产生。在其他情况下，瞬时值在经过多次重启后也要保证惟一性。有两种产生瞬时值的一般方法：第一种办法是使用一些装置中时钟的当前时间，这保证了你永远不会用两次同样的时间代码；第二种办法是使用在第10章中已详细讨论的PRNG。用随机瞬时值的缺点是该值要相当大。为了达到128比特的安全水平，我们需要使用256比特的随机瞬时值。并不是所有的装置都支持这样大的瞬时值，而且，PRNG很难在特定的平台上应用。因此，可靠的时钟才是产生瞬时值的有吸引力的方法。

17.1.3 单调性

时间的一个有用特性就是它一直在向前发展，从来不能停止或倒退。有一些密码协议用到了这个特性。在密码协议中包含时间会防止攻击者把旧的信息放到当前的协议中。毕竟，那些信息中编码的时间不在当前协议的时间跨度内。

时钟的另一个重要应用是审计和日志。在任何交易系统中，记录已经发生的事件都是很重要的。如果产生了争执，日志就会提供必需的数据来追踪确切的事件序列。在每次日志记录事件中包含时间是很重要的。如果没有时间戳，就很难知道哪些事件属于同一交易以及事件是以怎样的顺序发生的。由于同步的时钟彼此不会偏离（太多），时间戳可以允许在不同机器上的来自不同日志的事件发生关联。

17.1.4 实时交易

下一个例子来自于Niels的电子支付系统。为了支持实时支付，银行需要运行实时金融交易系统。要想完成审计日志，就要满足几个要求。首先，应该有一个清晰的交易序列。给定两个交易*A*和*B*，知道哪一个需要先完成是很重要的，因为一个交易的结果依赖于另一个是否已完成。记录这一序列的最简单的办法是给每个交易一个时间戳，这要在有可靠时钟的前提下才能完成。

一个不可靠的时钟也许会给出错误的时间。如果时钟偶尔向回移动的话，还不会有什么危害：当前时间应该比上次交易完成时的时间戳大，这是很容易检查的。然而，如果时钟向前移动的话，就会有问题了。假定半个小时的交易完成了，这时时钟设置为2020年。你不能去改变那些交易的时间戳，人工修改金融记录是不被接受的。你在2020年之前不能进行任何新的交易，因为那将改变由时间戳所决定的交易的顺序。对于这个问题有很多种解决办法，但有一个可靠的时钟肯定是最好的办法。

17.2 使用实时时钟

大多数台式计算机包含一个实时时钟芯片和一块小电池，这是安装在机器中的小的数字钟。就是通过它，你的机器在早晨启动时才知道是什么时间了。为什么不使用这个时钟时间呢？

实时时钟芯片对普通的应用来说是足够了，但在安全系统中，我们必须要求更高的标准。作为安全系统的一部分，即使敌人故意去操纵时钟，它也应该给出正确的时间来。第二个原因是时钟有时会出错。在一般的应用中，时钟出错会让人很生气，但还不是很危险。如果时钟是安全系统的一部分，那么它的错误能带来更大的伤害。

在一般硬件中的实时时钟并不像我们所需要的那样可靠和安全。在过去的十年里，我们已经亲自经历了几次实时时钟芯片的故障。而且，那些故障都是自发的，没有恶意攻击者去破坏它。大多数故障的原因都很简单：在一台旧机器上，由于电池电量不足，时钟就停止了或重新设置成1980年了。或是有一天你启动机器，时钟已经设置成2028年的某一天了。

除了实时时钟的意外错误，我们还不得不考虑主动的攻击。有些人也许要用某种方式操纵时钟。改变时钟时间是困难还是容易，这要由计算机的具体情况决定。在有些系统中只有特定的管理员才能改变时钟，而有些系统的时钟可以被任何人修改。

17.3 安全隐患

有几种类型的攻击与时钟相关。

17.3.1 时钟回置

假定攻击者能够把时钟设为过去的任何一个时间。这也许会导致各种产生伤害的行为，机器会错误地相信自己生活在过去。也许攻击者以前是一个临时雇员，可以访问一些数据，但这种访问权限已经过期了。由于时钟的错误，计算机也许能让这个非雇员访问敏感的数据。当用户被撤销了一些访问权限时，这种问题就发生了。把时钟往回设置也许可以恢复他的访问权限，当然，这还要取决于系统的其他部分是怎样设计的。

另一种有趣的攻击是关于自动完成的任务的。一个人力资源部门使用的计算机在每个月末自动地用银行的存款做工资支付。像这样自动完成的任务是通过一个程序触发的，这个程序检查时间和要完成任务的列表。重复地把时钟往回设置能够重复触发任务。如果任务被设置成午夜开始，那么攻击者把时钟设为23 : 55(11 : 55 pm)，然后等待任务的执行。当任务完成以后，他又一次把时钟调回。他一直在重复做这些，直到银行账上没有余额为止。

在金融系统中，另一个问题出现了。利息计算会根据交易完成的时间给出不同的结果，因此得到正确的交易时间是很重要的。如果你往信用卡上打入一大笔钱，你刚做的在线付款在银行计算机上显示的是6个月以前发生的，因此就会少付6个月的利息。

17.3.2 让时钟停止

每个设计者都会本能地认为时间是不会静止的。这是一个没有说出来的假定，因为这太明显了以至于不用给出说明。他们设计的系统也都是假定时间在正常地运行。但是如果时钟停止了,时间似乎就静止下来。这时情况就不一样了。而且很多系统将会以不可预测的方式来运行。

这种情况引起的简单问题是：在审计日志和报告里得到的是错误时间。交易的确切时间是相当重要的，把一份时间和日期都是错误的正式工作报告送出去，情况就会变得非常复杂。

在实时展示中其他问题也有可能发生。也许GUI程序员用一个简单的系统来展示实时交易的当前状况，每隔十秒用最近的数据更新一次展示。但由于存在各种延迟，并不是所有金融交易的报告都能以同一速度到达。仅报告收到的最新数据将会产生不一致的金融状况。也许一个交易的一部分已经报告了，但另一部分还没有报告。在股份还没有从股票账户撤出之前，银行的余额已经增加了。会计也不愿意得到一个还没有汇总在一起的报告。

因此程序员对此做了改进。每个金融交易的报告用时间戳标记，并且存放在本地的数据库中。为了展示一致的报告，他采用一个特定的时间点，报告那一段的金融状况。例如，如果最慢的系统在报告中有5秒的延迟，那么他就展示7秒前的金融状况。这虽然多了一些展示延迟，但保证了报告的一致性。时钟停止之前都是这样的。如果时钟停止，报告会一次又一次显示出同样的状况，即相对于（停止的）时钟7秒前的状况。

17.3.3 时钟前置

把时钟向前设置会使计算机认为自己生活在将来，这会引起简单的拒绝服务攻击。如果时钟被设成四年以后，所有的信用卡交易都会因为卡已到期而被拒绝。你因此也不能在线预定飞机票了，因为还没有那段时间的飞机时间表。

在eBay交易中，重大的竞标都发生在最后几秒。如果你能够把eBay的时钟向前移一点点，你就可以排除许多竞标者，以低价赢得这个项目。

我们的很多朋友在汇票系统中也会遇到问题。由于软件的毛病，时钟跳到30年后，因此汇票系统开始对所有的客户收缴30年未付的账。这种情况不会导致直接的经济损失，但客户如果使用了自动从银行账户或信用卡上借记的功能，那就不一样了。客户关系肯定不会搞得很好。

时钟向前设置也会带来一些直接的安全风险。有很多这样的情形：某些数据在特定的时间内是保密的，而此后可以公开。在自动系统中，时钟向前设置就可以使这些数据被访问了。如果有一个公司的分红备案，那么提前得到这个数据就会获利不匪。

17.4 产生可靠的时钟

对于时钟问题我们还没有一个简单的解决办法。我们可以提供一些建议和技术，但很多细节问题主要取决于具体的工作环境和风险分析，然后才能给出整体的方案。

大多数计算机在启动时有某种类型的计数器开始工作，这也许是CPU时钟循环的计数，也许是更新的计数器或其他类似的计数器。这个计数器能用来跟踪最后一次重启后的时间。它不是时钟，不能提供时间的信息，但它能用来测量事件之间花费的时间，只要这些事件是在最后一次重启后发生的。

计数器的主要作用（至少是与时钟问题相关的机制的主要作用）是检查实时时钟的突发性错误。如果时钟没有正常运行，就会显示出与计数器不一致。对其进行检查并对时钟的错误方式提出警告是很简单的。要注意的是，如果时钟的时间被一授权用户改变了，那么时钟时间和计数器值之间的关系也要被相应地修改。

第二种简单的检查办法是跟踪最后关机的时间，或是数据写入磁盘的最后时间。时钟不应该往回跳跃。如果你的计算机启动时是1980年，那很明显是出毛病了。这种办法还能防止时钟向前跳跃太多。大多数计算机至少一周启动一次。如果机器启动间隔超过一周，你就应该让用户确认正确的日期①。这样就能防止时钟向前跳跃一周以上。

此外还有其他一些检查时钟的办法。你可以询问Internet或intranet上的时间服务器，还有广泛应用的时间同步协议，像NTP[66]或SNTP[67]。这些协议中有一些甚至能提供时间数据的认证，使得攻击者不能篡改机器。当然，认证需要一些密钥基础设施。时间服务器的共享密钥可以是人工分配的对称密钥，但人工分配密钥还是有争议的。我们也可以用PKI来实现，但就

① 由于用户一般不看信息就敲OK键，因此让用户在不知道时钟时间的情况下写入当前时间是比较好的。

像第19章要讲到的，大多数PKI系统也需要一个时钟，这就导致了鸡生蛋还是蛋生鸡的问题。如果你要依靠时钟同步协议提供的密码保护，一定要小心。整个系统的安全性关键在于协议的安全性，而我们还没有看到关于这些协议的任何安全评论。

17.5 相同状态的问题

这是一个在一些硬件平台上存在的很严重的问题。我们这里要讨论小型嵌入式计算机——像门锁或智能卡解读器之类的东西，这些装置都含有一个小的CPU、小容量的RAM、存储程序的存储器、一些信道以及完成特定任务的硬件。

你会注意到，实时时钟没有包含在内。加入时钟需要有额外的芯片和振荡晶体，最重要的是还要有电池。就算不考虑额外的花费，加入电池也会使装置变得更复杂。你肯定要为电池是否有电感到担心。而且电池对温度波动是敏感的，一些电池中的有毒化学材料会带来装运硬件的问题。基于所有这些原因，很多小计算机没有实时时钟。

这样的小计算机在每次启动时都以同样的状态开始，它从同样的存储器中读取同样的程序，初始化硬件，然后开始操作。由于这是一本密码学的书，我们假定有一些密码协议用于同系统的其他部分通信。但有这样一个问题：没有时钟或硬件随机序列发生器，嵌入式系统将总是重复同样的行为。假定攻击者一直在等待机会，直到因为有一辆卡车要通过，看门的计算机要打开大门时，他就在大门刚要开启之前重启计算机（例如，瞬间地中断电力供应）。当一些初始化过程完成后，中央系统将通过通信信道命令计算机打开大门。第二天攻击者又一次重启计算机，并且送出与第一次完全一样的信息。由于这个看门的计算机以同样的状态开始，并且看到同样的输入，所以它就会做出同样的行为——把大门打开。这是很糟糕的。注意，如果看门的计算机使用一种时间同步协议，也不会起作用。协议信息能够与昨天的一致，计算机也没有办法检测到。相同状态的问题是不能通过任何协议来解决的。

实时时钟能够解决这个问题。小型嵌入式计算机能用一个固定的密钥加密当前时间来产生随机数据，这一数据反过来能用做密码协议的瞬时值。由于时钟从来不重复它的状态，嵌入式计算机也就可以避免掉进相同状态的陷阱里了。

硬件随机序列发生器也有相同的作用，它能够让嵌入式计算机在每次重启后有不同的状态。

但如果你没有实时时钟或随机序列发生器，就会遇到很多问题。有时可以从本地时钟振荡与网络计时的偏差中抽取一些随机性，但在较短的时间内从那里抽取足够的熵是很困难的。用10分钟来重启一个嵌入式计算机是不能接受的。

我们已经看到相同状态的问题一次又一次地出现。最极端的办法是，对这样的小计算机在做有用的工作之前改变硬件。这样的计算机很难卖给管理者，因为他们不想听到“某些事情是做不了的”这样的话。但是没有一个办法能让现有的不安全的系统变得安全。如果你没有从一开始把系统设计成安全的，那就几乎不会再得到安全了。

还有一种可能的解决办法，虽然它很少在实际中使用。有时可以在存储器中放置一个重启计数器。每当 CPU 重启时，存储器中计数器的数值将增加。这个办法存在很多问题。一些存储器只能更新几千次，如果一直更新计数器就会使机器坏掉。一些存储技术需要额外的电压来使其程序化，但这通常是不可能做到的。在一些设计中仅仅能在存储器中设置比特，或者擦除存储器中的所有存储数据。后面一种选择是不可能的，因为这样会失去机器中的主要程序。即便是所有这些问题都解决了，修改存储器和在机器的电力供应被掐断的情况下使得计数器一直保持增长也是很困难的，这种存储计数器的办法只在很少的情况下可以使用。

17.6　时间

当讨论时钟时，我们对如何选择时间有一些小建议。不要选择当地时间。当地时间是我们手表上的或其他钟表上的时间。当地时间是随着黎明时间的变化而改变的。这些改变就带来了一些问题：由于秋季时钟被往回设置一小时，有一些时间值每年都会重复，这就意味着时间不再是惟一的和单调的。当时钟在春季向前设置一小时后，有些时间值就不可能出现了。而且，黎明开始和结束的确切时间在不同国家是不相同的。在一些国家，时间制度每隔几年就会改变，你却不能因此而不断更新软件。那些带着膝上电脑旅行的人们要根据当地时间不断改变电脑的时间，这就使问题变得更严重了。

一个很明显的选择是使用 UTC 时间。这是一个自动时钟基础上的国际时钟标准，被广泛地应用在全世界。任何一个计算机都能追踪当地时间与 UTC 的偏差，以此来与用户交流。

UTC 存在这样一个问题：跳跃秒。为了让 UTC 与地球的旋转同步，大约每几年就有一个跳跃秒。由于所有的跳跃秒都是额外的秒，因此有的分钟会达到 61 秒。丢掉 1 秒在理论上是可实现的，这完全依赖于地球的旋转。问题是对计算机来说跳跃秒是不可预测的，把跳跃秒去掉会引起通过跳跃秒的时间间隔的测量不准确。这不是密码学的问题，但如果你想要一个很准的时钟，也许就要把问题考虑清楚。所有的计算机软件总是假定每分钟有 60 秒。如果你直接与一个 UTC 时钟同步，那么跳跃秒的插入就会带来很多问题。有可能你的时钟有 1 秒是重复的。这虽是个小问题，但时间值的惟一性和单调性又一次被破坏了。

对于大多数应用来说，时间戳的单调性和惟一性比时钟的完全同步更重要。只要能保证时钟永远不会往回跳一个跳跃秒，怎样来解决这个问题并不重要。

关于 UTC、GMT、TAI 和 UT1 等不同的时间标准的更多信息都可以在因特网上找到。

17.7　结论

很遗憾，我们并没有给出一个很理想的解决办法。产生一个可靠的时钟是很重要的，尤其是假定有很多恶意攻击者时。最好的解决办法还是要根据具体情况来决定。祝你好运！

第18章　密钥服务器

最后我们来讨论密钥管理。毫无疑问，这是密码系统中最困难的问题，这也是我们把它放在最后的原因。我们已经讨论了怎样加密和认证数据，以及怎样协商双方的共享密钥。现在我们需要找到一个办法，让 Alice 和 Bob 在因特网上彼此认识。你将会看到，这会变得很复杂。密钥管理是尤其困难的，因为它考虑到了人，而不单纯是数学，人是更难理解和预测的。

在开始之前，我们要搞清楚一件事。我们仅仅从密码学角度讨论密钥管理，而不是从组织的角度。组织的角度应该包括把密钥发行给谁、密钥能访问哪些资源、怎样验证拿到密钥的人的身份、存储密钥的安全策略、与这些规则紧密相连的验证机制等。每个组织会根据不同的需求以及现存的组织结构来考虑这些问题，我们所关心的仅仅是直接影响密码系统的那部分。

密钥管理的一个办法是让一个可信实体来分发所有的密钥，我们把这个实体叫做密钥服务器。

18.1　基本概念

基本概念是很简单的。我们假定每个人都与密钥服务器建立一个共享密钥。例如，Alice产生一个只有她和密钥服务器知道的密钥K_A，Bob也建立一个只有他和密钥服务器知道的密钥K_B。其他人都以同样的方式建立密钥。

现在假定 Alice 想与 Bob 进行通信。她没有与 Bob 通信的密钥，但她可以与密钥服务器安全地通信。密钥服务器反过来也能与Bob安全地通信。我们能简单地把所有信息传给密钥服务器，把它当成一个巨大的邮局。但对密钥服务器来说存在一些困难，它将不得不处理大量的信息。一个更好的解决办法是，让密钥匙服务器建立一个 Alice 和 Bob 的共享密钥 K_{AB}。

18.2　Kerberos

这就是Kerberos——一个广泛应用的密钥管理系统[58]——产生的基本想法。Kerberos是建立在 Needham-Schroeder 协议[76]基础上的。

下面我们从基础的层面上看看 Kerberos 是怎样工作的。当 Alice 想与 Bob 对话时，她首先与密钥服务器联系。密钥服务器向Alice发送一个新的密钥K_{AB}和用Bob的密钥K_B加密后的K_{AB}。这些信息都用K_A加密，因此只有 Alice 能读。Alice 把用K_B加密的信息发送给 Bob，Bob 对它解密，然后得到K_{AB}，这是只有 Alice 和 Bob 知道的会话密钥——当然，密钥服务器也知道。

Kerberos 的一个特点是密钥服务器（在 Kerberos 术语中叫做 KDC）不必经常更新状态。当然，密钥服务器要记住与每个用户共享的密钥。但当 Alice 请求 KDC 建立一个她和 Bob 之间的密钥时，KDC 完成这个任务后，把结果扔掉。它并不跟踪用户之间建立了哪些密钥。这是一个很好的性质，因为它可以让密钥服务器以一种简单的方式把重负分布在几个机器上。由于没有状态要更新，Alice 在这一刻可用密钥服务器的一个副本进行交谈，而下一刻用另一个副本。

事实证明，Kerberos 系统所需的密码协议是很复杂的。刚开始设计这样的协议看起来很容易，但即使是经验丰富的密码学家提出的方案后来也都被攻破了。存在的那些缺陷都是很微妙的。我们这里不去解释这些协议，我们甚至都不愿意设计这种类型的协议。如果你想使用一种这样的协议，要用 Kerberos 最后的版本。它已经存在很长时间了，并且被很多专家研究过。

18.3　更简单的方案

有时使用 Kerberos 是不可能的。协议太复杂，而且还有很多限制。服务器不得不存储所有用户的密钥，每个用户还需要一个可靠的时钟。这些要求有时候是无法满足的。

如果我们不太强调效率，那么就能给出一个更简单更强健的方案来。事实证明它是很有用的，允许密钥服务器维持状态。当代的计算机与刚开始设计 Kerberos 的那个年代的机器相比要强大多了，它们可以没有任何困难地维持上万个用户的状态。甚至对有 10 万个用户的大系统也不成问题：如果每个用户在密钥服务器中需要占 1 KB，那么存储所有的状态仅需要 1 亿字节的内存。密钥服务器需要很快地建立所有请求的密钥，但对现代的快速的计算机来说也是很容易实现的。

我们这里仅讨论有一个密钥服务器的情况。有一些技术可以用来把密钥服务器的状态分布在几个机器上，对此我们将不进行详细的讨论，因为你并不真想用一个有上万用户的密钥服务器，那样太冒险了。大的密钥服务器的风险在于，所有的密钥都放在一个地方，使得密钥服务器成为一个很容易攻击的目标。密钥服务器也必须一直在线，这就意味着攻击者能随时与它进行通信。当前的状态并不能保护计算机免受网络的攻击，而把密钥都放在一个地方就会引起灾难。对小型系统来说，被密钥服务器保存的密钥值小，因此威胁也减少了[①]。在后面几章里，我们要讨论适合大系统的密钥管理系统的更好解决办法，我们这里仅讨论小系统的密钥服务器——不超过几千个用户。

18.3.1　安全连接

下面是我们这个简单方案的简短阐述。首先，我们假定 Alice 和密钥服务器共享密钥 K_A。我们不直接使用这个密钥，而是用它来运行一个密钥协商协议，就像在第 15 章讨论过的那样。

① 我们并不想在系统中留有隐患，但在密钥管理中，最终总是要给出一个折中的方案来。

（如果K_A是一个口令，那么最好使用在15.12节中讨论的适用于低熵口令的协议，但那会牵涉到专利问题。）密钥协商协议建立了密钥服务器和Alice之间的新密钥K'_A。所有其他用户也都与密钥服务器运行同样的协议，建立新的密钥。

Alice和密钥服务器用K'_A来产生一个安全的通信信道（详细内容见第8章）。通过使用这个安全的信道，他们可以进行安全的通信。私密性、可认证性、重放保护等特性都能通过信道来提供，所有进一步的通信都发生在这个安全的信道上。其他的用户也都与密钥服务器建立了类似的安全信道。

18.3.2 建立密钥

现在设计一个协议来建立Alice和Bob之间的密钥就更容易了。我们只需要考虑信息丢失或是被攻击者删除的情况，因为安全信道已经防止了其他类型的攻击。这样一来协议就相当简单了。Alice请求密钥服务器来建立她和Bob之间的密钥，密钥服务器把新密钥K_{AB}发送给Alice和Bob作为答复。密钥服务器甚至可以通过Alice把信息传递给Bob，使得它无需直接与Bob通信。

这样做对系统有一个限制：在Alice请求密钥服务器建立同Bob的共享密钥之前，Bob的密钥服务器必须运行密钥协商协议。这是否是一个问题还要由具体的环境来决定，可能的解决办法也是一样的。

18.3.3 重新生成密钥

像所有密钥一样，K'_A必须有一个有限的使用期。这是很容易做到的，因为Alice总能重新运行密钥协商协议（用原始密钥K_A做认证）来建立新的密钥K'_A。一个密钥有几个小时的使用期限似乎在大多数情况下是合理的。

因为我们总能够重新生成密钥，密钥服务器不必以可靠的方式来存储安全通信状态。假定密钥服务器出现事故，失去了所有的状态信息，而只要它还记得K_A（以及其他用户相应的密钥）就没有问题。我们能在密钥服务器与每个用户之间重新运行密钥协商协议，因此，虽然密钥服务器不是无状态的，但当运行协议时它不必修改自己的长期状态——储存在不易丢失的媒介上的那一部分。

18.3.4 其他性质

也许从应用的角度来说我们的方法并不比Kerberos简单，但从理论上看确实简单了。这种安全信道使得抵抗协议的可能攻击更容易了。使用密钥协商协议和我们已经设计好的安全信道就是模块化是如何帮助设计密码协议的一个很好的例子。

用密钥协商协议来建立安全的信道还有另外一个优势：我们能得到前向的安全性。如果Alice的密钥K_A今天被泄露了，那么她的旧密钥K'_A不会被揭示，因此所有过去的通信仍然是安全的。

在本书前面的章节中，我们给出了密码函数设计的详细例子。密码学是一直向前发展的，我们阐述的一个密钥服务系统有可能在将来被淘汰。如何设计密钥管理系统应该说是收集特定应用的具体需求并且得到完好用户界面的问题，而不是密码学的问题。为了在这里能够解释例子的设计选择，我们将不得不设计和说明整体的环境和组织结构、构成危险的环境以及需要密钥管理的应用。这些更应该是小说体密码学书籍的内容，而不是应用密码学书籍的内容。

18.4　选择

如果你想要应用一个中心的密钥服务器，可能的话应使用Kerberos。它是可行的，并且是被广泛使用的。

在不适合 Kerberos 的情况下，你将不得不设计和应用一些类似于我们已经阐述的解决办法，但如何设计密钥服务系统应该是很主要的操作。对于大多数我们已经看到的密码学应用，在密钥服务系统上所花费的时间至少应该与整个应用一样多。

第19章　PKI之梦

本章我们将介绍PKI并说明它是怎样解决密钥管理问题的。掌握这些知识是很重要的，下一章我们将讨论为什么PKI在实际中不能很好地应用，但本章我们将只考虑理想情况，即PKI能解决所有的问题的情况。

19.1　PKI的简短回顾

PKI是一个公钥基础设施（Public-Key Infrastructure），它是让你知道哪个公钥是属于谁的基础设施。其标准的阐述如下。

有一个中央权力机构叫Certificate Authority，简称CA。CA有一公/私密钥对（例如，RSA密钥对），并且发布了它的公钥。我们假定每个人都知道CA的公钥。由于这个公钥保持长时间不变，这是很容易做到的。

为了加入到PKI，Alice生成了她自己的公/私密钥对。她的私钥是秘密的，把公钥PK_A传给CA，对它说："嗨，我是Alice，PK_A是我的公钥。"CA证实Alice的身份后，签署一个类似于"密钥PK_A属于Alice"的数字声明。这个签署的声明就叫做证书，它证实密钥属于Alice。

如果Alice想要与Bob通信，她就把自己的公钥和证书发送给Bob。Bob有CA的公钥，因此他能证实证书的签名。只要Bob相信CA，他就会相信PK_A确实属于Alice。

通过同样的过程，Bob得到由CA签署的他的公钥证书，并且把公钥和证书传给Alice。他们现在就彼此知道对方的公钥了。这些公钥反过来可用于运行密钥协商协议，以建立安全通信的会话密钥。

我们现在需要的就是使每个用户都信任的CA。每个用户需要得到有证书的公钥，并且还需要知道CA的公钥。这样一来，每个用户都可以同其他所有用户安全地进行通信了。

这些看起来是相当简单的。

19.2　PKI的例子

为了使本章后面的部分更容易理解，我们首先给出如何应用PKI的例子。

19.2.1　万能的PKI

我们的最终梦想就是拥有一个万能的PKI。它是一个像邮局一样的大组织，给每个用户发行公钥证书。它的好处就是每个用户仅仅需要得到一次证书，因为同样的公钥可以用在每个应

用中。因为每个人都信任邮局，或相信成为通用CA的其他组织，这样所有的人就都可以安全地互相通信了。

如果说我们的阐述有些像传说，那是因为它确实如此。从来就没有万能的PKI，而且永远也不会有。

19.2.2 进入 VPN

这里给出一个更实际一些的例子。有VPN的公司允许它的雇员从家里或是旅行时从宾馆进入公司的网络。VPN的进入点必须能识别要进入网络的人以及他们有何种确切的进入级别。公司的IT部门扮演CA的角色，给每个雇员一个证书，使得VPN的进入点能够识别雇员。

19.2.3 电子银行

银行允许它的客户在银行网上进行金融交易。在应用中正确地识别客户的身份是至关重要的，就像在法庭上能提供可接受的证据一样重要。银行本身就是CA，它能证实客户的公钥。

19.2.4 提炼传感器

提炼的复杂性是很高的。在管子和接口之间放置的是上百个传感器，可以测量温度、流率和压力之类的指标。在提炼过程中篡改传感器数据是一种很严重的攻击，把错误的传感器数据发送到控制室从而导致发生大爆炸也许并不是很困难的。因此，控制室得到正确的数据是很必要的。我们可以用标准的认证技术来保证传感器数据没有被篡改，但是为了确认数据确实是来自传感器，就需要某种密钥基础设施。公司可以作为CA，对所有的传感器建立PKI，使得每个传感器都能被控制室识别。

19.2.5 信用卡组织

信用卡组织是遍布世界的几千个银行之间带有风险的合作组织。所有这些银行必须能够交换支付，因为在银行*A*有信用卡的用户必须能把钱付给在银行*B*开户的商人。银行*A*与银行*B*需要以某种方式处理这一交易，而且要保证安全。PKI就可以让所有的银行彼此识别身份以进行安全的交易。在这种情形下，信用卡组织扮做CA，给每个银行发行证书。

19.3 其他细节

在实际生活中情况更复杂，因此常常用到PKI方案的各种扩展形式。

19.3.1 多层证书

在很多情况中，CA被分成多个层。例如，中心信用卡组织不直接给每个银行发行证书。相反，他们有区域性机构来管理银行。这样我们就得到了一个两层的证书结构。中心CA签署

PDG

一个关于区域CA公钥的证书，就好像说："密钥PK_X属于区域机构X，它可以用来证实其他密钥"。每个区域机构能够给单个的银行密钥发行证书。银行的密钥证书包含两个签名的消息：中心CA证实区域机构密钥的授权消息和区域机构关于银行密钥的证书。这样的结构叫做证书链，这样的链可以扩展到任意多的层。

这样的多层证书结构是很有用的。它基本上是把CA的功能分层，这对大多数组织来说是很容易处理的。几乎所有的PKI系统都有多层结构。这种结构的一个缺陷是证书太大，验证需要更多的计算。这种结构的另一个缺陷就是加入系统的每个CA都会带来新的攻击点，因此降低了整个系统的安全性。

对于多层证书结构的这些缺陷，一个解决办法是打破证书的层次，这种办法在实践中还没有见到过。继续我们上面的例子，一旦银行有了两层的证书，它就把证书送到中心CA。中心CA对两层证书做验证，然后用CA主密钥签署一个银行密钥的单个证书。一旦层次像这样被打破，加入额外层次的代价就会变得很小。但是加入额外层次也许并不是一个好主意，多层结构的效率并不是很高。

在处理这样的证书链时一定要小心，他们加入了更多的复杂性，而复杂性总是不好的。下面是最近发生的一个例子：Internet上的安全站点采用PKI系统以允许浏览器识别正确的网址。实际上这个系统不是很安全，因为大多数用户不能证实他们正在使用的网址的名字。去年，一个致命的bug出现在所有微软操作系统中证实证书的库里。证书链的每个元素包含一个标识，用它来说明元素所证实的密钥是否为CA密钥。CA密钥可以证实其他的密钥，非CA密钥不能证实其他的密钥。这是一个重要的区别。遗憾的是，有问题的库不能检查这个标识。因此攻击者能够购买域nastyattacker.com的证书，用它来给amazon.com签署证书。微软IE使用了这个有错误的库，它将接受nastyattacker.com关于伪造Amazon密钥的证书，像真的一样来显示错误的Amazon网址。这样一来，花费很多精力构建的世界范围的安全系统就被一个库里的一个简单的小bug完全攻破了。一旦这个bug被发布，相应的补丁也会出现（从而需要多次努力修改所有的漏洞），这是一个小bug毁掉整个系统安全的好例子。毫无疑问，现在仍有人还在使用没打补丁的版本。

19.3.2 有效期

没有一种密码学密钥能被无限期地使用，因为总是存在着密钥被攻破的风险。规则的密钥更换能让你从密钥的泄露中恢复出来，虽然这很慢。一个证书也不能是永远有效的，因为CA密钥和被证实的公钥都有截止期。即使不考虑密码学方面的原因，有效期对于保持信息的更新也是很重要的。当一个证书到期时，新的证书就会被重新发布，这就为更新证书的信息制造了机会。一般的有效期是在几个月到几年之间。

几乎所有的证书系统都包含截止的日期和时间。没有人会接受一个过期的证书。这就是PKI用户需要时钟的原因。

很多证书的设计还包含其他数据。通常，证书中除了有截止期，还有有效的起始期。有的还包含证书的不同类别、证书序列号、发行日期和时间等。这些数据中有些是有用的，有些是无用的。

证书最通用的模式是X.509 v3，它包含了大量过时的东西。如果你要干蠢事的话，就看看Peter Gutmann的X.509模式导引[40]。如果你工作的系统不需要与其他系统协作的话，最好忘记X.509。如果必须要用X.509，我们只能对你表示同情了。

19.3.3 独立的注册机构

有时，你会看到有的系统包含一个独立的注册机构。这是一个行政问题。公司里只有HR部门才能决定谁是雇员。但IT部门要运行CA，这是一个技术工作，他们不会允许HR部门来完成。

有两种好的解决办法。第一种是使用多层证书结构，让HR部门成为子CA。这就自动地提供了必需的灵活性来支持多机构。第二种办法与第一种很相似，只是一旦用户有了两层证书，他就能从中心CA得到一层的证书。只要通过把两个消息协议加入系统，就排除了每次使用时检查两层证书的负担。

一个不太好的解决方案是把第三方加入到密码协议中。项目说明中会具体讨论CA和可以叫做RA（注册机构）的另一方。CA和RA被看做完全独立的实体，这会把几百页的说明加到系统中。这种办法是很糟糕的。于是有必要说明RA和CA的关系。我们甚至看到过RA授权CA发行证书的三方协议。这种想法是很疯狂的，但这是把用户需求施加到技术解决方案上的一个好例子。用户需求仅仅说明了系统的外部行为，公司的HR部门和IT部门需要有独立的功能。但这并不意味着软件对HR部门和IT部门有不同的代码。在很多情形下，两个部门使用的功能大多相同，使用的代码也大多相同。使用一个证书功能的集合比包含CA和RA这种直接建立在原始需求上的设计更简单、更有效、更灵活。一个两层的CA方案允许HR和IT共享大多数相同的代码和协议。在这种情形下，差别主要是用户界面，这应该是很容易实现的。这也许仅需要几百行额外的代码，而不是能折合成上万行代码的几百页的说明。

19.4 结论

我们描绘的是一个梦想，但它是一个很重要的梦想。对于大多数行业来说，PKI是密钥管理的第一个用语，也是最后一个。人们已经被灌输了这个梦想，以至把它看做是很自然的事物。为了能够理解他们，就必须理解PKI梦想，因为他们说的很多东西都在这个梦的范围之内。人们可能认为有办法来解决密钥管理问题，这种感觉是如此的美妙……

第20章　PKI的现实问题

时间能毁灭梦想。PKI的基本想法中存在着一些问题，但不是在理论上。理论与实践是不同的，PKI在真实的世界中并不能像在梦想中一样简单地工作，这就是PKI经过几年的大力宣传也没有与现实相结合的原因。

20.1　名字

我们以一个相对简单的问题来开始讨论：名字的概念。PKI把Alice的公钥与她的名字绑定在一起。那么名字是什么呢?

我们以一个简单的背景来开始。在一个小村庄里，每个人都互相认识。每个人都有一个名字，这个名字或是惟一的，或是使它为惟一的。如果有两个人叫John，他们会很快被称为大John和小John或类似的名字。每个名字对应一个人，但一个人也许有几个名字，大John可能也叫做Sheriff。

我们这里谈论的名字不是出现在合法文件中的名字，而是人们提到你时用的名字。名字是用来指示某个人的任何一种数据，或更一般地说，是指一个实体。你"正式"的名字仅仅是很多名字中的一个，对很多人来说是很少被用到的一个。

由于村庄变成了一个城镇，居民的数目增长了，以至彼此并不都认识。名字开始失去了与人的直接关系。在城镇中也许仅有一个J. Smith，但你并不认识他，名字与实际的人产生了分离。你开始谈论你实际上从来没有遇到的人。也许你在酒吧里谈论富人Smith先生，而他刚刚离开这儿。两周后你发现这个Smith先生正是两个月前参加你的篮球队的那个人，你叫他John。毕竟一个人可以有多个名字，某人叫哪个名字以及某个名字是指哪一个人并不是很明显的。

随着城镇扩大为城市，这种变化就更明显了。很快你就只认识一小部分人了。而且，名字也不再是惟一的了。如果城市中有100个叫John Smith的人，名字就真的不能帮助你来找到他了。名字的含义开始与背景有关。Alice也许认识3个John，但当她在工作中谈论"John"时，从背景中看很明显是指楼上销售部的John，在家谈论时也许是指邻居的小伙子John。名字和人之间的关系变得更模糊了。

现在来考虑Internet，在网上很快就会有100万人在线。在那里"John Smith"意味着什么意义呢？几乎什么意义都没有：有太多人叫这个名字了，因此我们用E-mail地址来取代传统的名字。你现在可以与jsmith533@yahoo.com联系，那肯定是惟一的一个名字，但实际上你并不能把它与你曾经遇见的某人联系到一起。即便你能找到他的地址和电话之类的信息，他也很有

可能生活在地球的另一边。你永远不会遇见他，除非你设法去做这件事。毫无疑问，人们一般都呈现不同的在线个性，总是有很多个名字。大多数用户在一段时间后会获取多个E-mail地址（我们现在就有十多个）。但判断两个 E-mail 地址是否属于同一个人是相当困难的。更复杂的是，有一些人共享一个 E-mail 地址，因此“名字”就是指他们所有人。

有一些大型组织试图给每个人分配名字，最广为人知的就是政府机构。大多数国家要求每个人有一个官方名字，该名字被用在护照和其他官方文件中。因为名字本身不是惟一的——很多人有相同的名字，因此实际上它常常用地址、驾照号和生日之类的加以扩展。然而，这仍然不能保证一个人的惟一身份[①]。而且，有一些标识在人的一生中还会有变化。人们会改变他们的地址、驾照号、名字甚至性别。只有生日不能改变，但是有许多人谎报他们的生日，实际上也就相当于改变了它。

如果你认为每个人都应该有专一的政府批准的官方名字，这也不太现实。有些人是居无定所的，他们根本没有官方名字；有些人有双重国籍，两个政府都给他一个官方名字——基于各种原因，官方名字并不能达成一致。两个政府也许使用不同的字母表，因此名字是不一样的。一些国家要求采用适合本国语言的名字，会把外国名字用自己的语言修正为类似的“正确”名字。

为了避免混淆，很多国家给每个人分配惟一的号码，像在美国是SSN，荷兰是SoFi。这个号码的完整意义是给每个人提供一个惟一的固定的名字，使得他的行为能被追踪并联系到一起。在很大程度上说，这些号码方案是成功的，但它们也存在着不足。分配的号码和实际人之间的联系并不是很紧密，假的号码在一定的经济活动中被大范围地使用。而且由于这些号码方案是基于不同国家的，它们并不能提供全球的覆盖或惟一性。

名字的其他方面也应得到关注。在欧洲，有保密法来限制一个组织能存储哪些个人信息。例如，一个超市不允许询问、存储 SSN 或 SoFi 号，这就限制了政府号码方案的使用。

那么在PKI中应该使用什么名字呢？因为许多人有很多不同的名字，这就是一个问题。也许 Alice 想要两个密钥，一个做业务，一个进行私人通信。她也许用少女时的名字做业务，用婚后的名字进行私人通信。如果你想要构建一个全球的PKI，像这样的事情很快就会导致严重的问题。这就是小范围特定应用的 PKI 比大型 PKI 工作得好的原因之一。

20.2　权力

谁是有权给名字分配密钥的CA呢？谁来决定用户的名字呢？谁来决定 Alice 是否为可访问 VPN 的雇员或银行客户呢？

对于我们的大多数例子来说，这是一个很容易回答的问题。老板知道谁是雇员，谁不是雇员；银行知道谁是客户。这就给出了哪个组织应该是CA的一种暗示。遗憾的是，对于全球PKI来说，似乎并没有一个权力的来源。这就是全球 PKI 不能正常工作的原因之一。

① 驾照号码是惟一的，但并不是每个人都有。

PDG

只要计划一个PKI，都不得不考虑由谁来发行证书。例如，公司有权决定谁是它的雇员。公司不能决定雇员的名字是什么，但它确实知道雇员在公司范围内的名字。如果“Fred Smith”的官方名字叫做Alfred，这并没关系。名字“Fred Smith”在公司内部就是一个好名字。

20.3 信任度

密钥管理是密码学中最困难的问题，PKI系统是解决这个问题的最好的工具之一。但是每件事情都依赖于PKI的安全性，因此也要依靠CA的可信赖性。想一想如果CA开始伪造证书，那会带来多少危害？CA能够伪造系统中的任何人，从而安全性完全被破坏了。

全球的PKI是很诱人的，但可信任度是它的问题所在。如果你是一个银行，你需要同客户联系，你会相信在世界另一边的.com吗？或是相信当地的政府官僚机构吗？如果CA犯了一些相当可怕的错误，你会损失多少钱？CA又愿意承担多少责任？这些都是很大的问题。如果CA的私钥被发布在网上，我们来想像一下产生的危害吧……

我们以传统的方式来考虑这个问题。CA是把密钥分发给大楼的一个组织，大多数大办公楼都有门卫，大部分门卫都是从外边的安全服务部门雇佣的。门卫只是进行验证：一个很直接的工作。但是决定谁得到哪个密钥并不是能指派另一个公司去完成的事情，因为它是安全策略的基本部分。基于同样的原因，CA功能不应向外分配。

世界上不存在被每个人都信任的组织，甚至也没有一个被大多数人信任的组织，因此永远不会有全球的PKI。按这个逻辑得出的结论就是，我们将不得不用很多小型的PKI，这就是我们建议的解决办法。银行可以是它自己的CA。毕竟，银行信任它自己，所有的客户也相信他们存钱的银行。对VPN来说，公司是它自己的CA，信用卡组织也能运行它自己的CA。

这里一个有趣的发现是，被CA使用的信任关系都是已经存在的，而且是建立在契约关系基础上的。当你设计密码系统时，总应该是这样的：所建立的基本信任关系都是基于已有的契约关系之上的。

20.4 间接授权

现在我们来考虑一般的PKI所遇到的大问题。PKI把名字和密钥捆绑在一起，但大多数系统对人的名字不感兴趣。银行系统想要知道授权了哪个交易，VPN想知道哪个地址允许进入。这些系统都不关心密钥属于谁，只是想知道密钥持有者被授权做什么。

最终，大多数系统使用某种访问控制列表（ACL），这是一个谁被授权做什么事的数据库。有时它通过用户来分类（例如，Bob被允许做下面的事情：访问目录/usr/bob中的文件、使用办公室打印机、访问文件服务器），但大多数系统让数据库通过行为来索引（对这个账户收费必须由Bob或Betty授权）。通常我们可以产生群组来使ACL更简单，但基本的功能是一样的。

现在我们有 3 个不同的对象：密钥、名字、做某事的许可权。系统想要知道的是哪个密钥被授权了哪种行为，换句话说，特定的密钥是否有特定的许可权。一般的PKI通过把名字和密钥绑定以及用 ACL 把名字和许可权绑定来解决这一问题。这是一种迂回的办法，它导致了额外的攻击点[28]。

第一个攻击点是 PKI 提供的名字 - 密钥证书，第二个攻击点是绑定名字和许可权的 ACL 数据库，第三个攻击点是名字的混淆：名字是很含混的，怎样来比较ACL中的名字是否与PKI证书中的名字是一样的呢？又怎样来避免给两个人赋予同样的名字呢？

如果对这种情况进行分析，就会清晰地看到技术设计是完全根据原始需求而来的。人们考虑的问题是识别密钥持有者的身份以及谁能够进入——这是一个门卫考虑的问题，自动系统可以用一个更直接的办法——门锁并不关心谁拿着钥匙，它允许任何拿钥匙的人进入。

20.5 直接授权

一个更好的解决办法是直接用PKI把密钥与许可权绑定，证书不再连接密钥和名字，它连接密钥和许可集合[28]。

那么所有使用PKI证书的系统现在就能够直接决定是否允许用户进入了。他们只看所提供的证书和密钥是否有相应的许可权，这既直接，又简单。

直接的授权从授权过程中去掉了 ACL 和名字，因此排除了这些点的攻击。当然，在证书发行时一些问题会重新出现。某些人必须决定允许谁做什么，并保证这个决定正确地被编码在证书中。所有这些决定的数据库和 ACL 数据库是等价的，但这个数据库更不容易被攻击。首先，去掉中心的 ACL 数据库以及相应的攻击点，很容易来分配做决定的人。这些做决定的人不需要进一步的安全设施就可以给用户发行相应的证书。这也改变了对名字的依赖，因为做决定的人都在层次结构的较低部分，他们只需要管理一小部分人。他们通常都了解用户，或至少是能看见用户，这在很大程度上避免了名字混淆问题。

那么我们可以在证书中去掉名字吗？

答案是否定的。虽然在正常的操作中不需要名字，但我们还要给审计核算提供日志数据。假定银行做了工资支付，这项操作是由对那个账户有权限的四个密钥之一授权完成的。三天后，CFO 给银行打电话询问做这笔支付的原因。银行知道支付被授权了，但它要提供给 CFO 更详细的信息，而不能是看起来比较随机的几千比特的公钥数据，这就是我们在证书中仍然包含名字的原因。银行现在可以告诉CFO这项支付是由属于“J. Smith”的密钥授权的，CFO就完全可以领会了。但重要的是名字仅仅对人才是有含义的，计算机从来不会算出是否两个名字是相同的，或名字属于哪个人。人更善于处理含混的名字，而计算机的优势在于处理像许可权设置这样简单的并且给出详细说明的操作。

20.6 信用系统

如果把这个原理再深入一步，就会得到一个很好的信用系统。这就是密码学家所说的超级PKI。基本的要求是，对于你做的每件事情要以签署证书的形式给出凭证。如果Alice有允许她读和写一个特定文件的凭证，她就能部分或全部授权给Bob。例如，她可以用Bob的公钥签署一个证书，上面标明“密钥 PK_{Bob} 通过 PK_{Alice} 的授权可以读文件 X”。如果Bob想要读文件 X，他就要提交这个证书和Alice有权限读文件 X 的证明。

信用系统能加入额外的特征。Alice能通过在证书中包含有效期来限制授权的有效期。Alice也可以限制Bob，使其不能授权读文件 X ①。

理论上说，信用系统是相当有效和灵活的。实际上它们却很少被使用。主要有以下几个原因。

首先，信用系统是很复杂的，开销过大。访问一个资源的授权也许要依靠半打的证书链来完成，而且每个证书都要被传递和检查。

第二个问题是信用系统会引起访问权限的细化管理。我们很容易把权限分成越来越小的部分，使得用户最终要花掉很多时间来决定分配给一个同事多少权限。这很浪费时间，但更大的问题是，当同事最终证实自己没有做这项工作的权限时，他也浪费了大量时间。也许这种细化管理问题能通过更好的用户教育和更友好的用户界面来解决，但那仍是一个公开问题。一些用户通过把（几乎）所有的权限都授权给需要访问的任何人来避免细化管理问题，但这却破坏了整个的系统安全。

第三个问题是需要开发一种授权语言。授权信息要以一种计算机能理解的逻辑语言来编写。这种语言要足够强大，以便能表示出所有想要的功能，然而又要相当简单，能快速得出结论链。它还应该是可证明的。一旦开发了信用系统，每个程序将包含相应代码来说明授权语言。把这样的语言升级到一个新版本是非常困难的，尤其是安全功能遍及到系统的每个角落。然而，要设计一种授权语言，使其具有足够的通用性以至能够满足所有将来的需求，这是完全不可能的，因为我们永远不会知道将来是什么样。这个问题有待进一步研究。

信用系统的第四个问题可能是不可克服的。具体的授权对普通用户来说是相当复杂的原理，似乎没有一种表明访问规则的方式能被用户理解。让用户做决定来分配授权是注定要失败的，我们在现实生活中已经看到过这种例子：在学生公寓里，一个人去ATM为几个人提现钞是很常见的一件事。其他学生借给他ATM卡和PIN号。这是一件很愚蠢的事情，然而社会上一些被认为是聪明的人还在做这种事。我们已经参观了很多公司，有时由于工作的原因会进人他们的局域网。我们非常吃惊地看到我们有那么多访问权限。当我们只需要看一两个文件时，系统管理员给了我们访问所有研究数据的权限。如果系统管理员没有给这个权限，普通用户当然不会有。

① 这是一个经常要用到的特征，但我们认为它也许并不是很好的。限制Bob的授权能力就会让他运行代理程序，通过其他人的凭证访问资源。这样的代理程序破坏了安全构架，应该被废弃，但如果没有其他原因来运行代理程序，这是仅有的能够成立的原因。总是存在人们需要授权的原因。

作为密码学工作者，如果用户能够管理好信用系统的复杂性，我们还是非常喜欢信用系统的想法的。毫无疑问，还有很多有趣的人与安全系统交互方面的研究工作要去做。

然而，有一个区域信用权限还是很有用的，也是必须的。如果使用分层的CA结构，中心CA签署关于子CA密钥的证书。如果这些证书不包含任何限制，那么每个子CA就都有无限的权力。这是很差的安全设计，我们只是增加了很多地方来存储重要的系统密钥。

在分层的CA结构中，子CA的权力应该通过证书来给予限制，这就要求有一种类似于授权的语言进行CA操作。究竟需要什么类型的限制还要视具体应用而定。需要考虑的是想要生成什么样的子 CA 以及这些子 CA 的权力需要怎样被限制。

20.7　修正的梦想

让我们概括一下已经提到的PKI的缺陷，然后给出一些修改设想。这是PKI更现实的表示。

首先，每个应用都有自己的 PKI 和自己的 CA。我们的世界包含了大量的小范围 PKI，每个用户同时是许多不同 PKI 的成员。

用户对每个PKI必须使用不同的密钥，因为在系统设计时系统之间没有经过协调，不能在不同的系统中使用相同的密钥，因此用户需要存储很多密钥，要占用上万字节的存储空间。

PKI 的主要目的就是把密钥同许可权绑定在一起。银行的 PKI 把 Alice 的密钥与可以进入 Alice账户的许可权联系起来。公司的PKI把Alice的密钥与可访问VPN的许可权绑定在一起。用户许可权的重大改变会要求发行新的证书。证书中仍然要包含用户的名字，但这主要是为了管理和审计的目的。

这种修正后的梦想更现实了。它比原来的梦更有效、更灵活、也更安全。这很容易使人相信它能够解决密钥管理问题。但下一节我们将会面临最困难的问题——永远都不能完全解决它，总是需要做一些折中。

20.8　撤销

在PKI中，最困难的问题就是撤销。有时一个证书不得不被撤销。也许Bob的计算机被别人使用过，从而泄露了他的私钥，也许 Alice 被调到另一个部门或者被公司解雇了，等等。总之，你可以想到需要撤销证书的许多种情况。

问题是证书仅仅是一串比特。这些比特已经在很多地方使用，而且储存在很多地方。无论如何努力，都不能让人们忘掉这个证书。Bruce 大约在十年前丢失了 PGP 密钥，他现在仍然会收到用相应证书加密的E-mail[①]。试图让所有人忘记这个证书是很不现实的。如果一个小偷进入 Bob 的计算机，偷走了他的私钥，可以肯定这个小偷也会对相应公钥的证书做一个副本。

每个系统都有它自己的需求，但总体上说，撤销需求是由 3 个变量决定的：

① PGP 本身有着类似 PKI 的结构，叫信用网。实际上，它太复杂了以至于不能向用户解释，它仅仅在小范围内应用。

- **撤销的速度**：撤销指令发出与最后一次使用证书之间所允许的最大间隔时间是多少？
- **撤销的可靠性**：在某些情况下撤销不是完全有效的，这可以接受吗？什么样的风险是可以接受的呢？
- **撤销的数目**：撤销系统一次能处理多少撤销请求？

对于撤销问题有两种可行的解决办法：撤销列表和短的有效期。

20.8.1 撤销列表

撤销列表（或CRL）是一个包含撤销证书列表的数据库，每个想验证证书的人都必须检查CRL数据库，看看那个证书是否已经被撤销。

中心CRL数据库有很好的性质，撤销几乎是瞬时完成的。一旦证书被加入CRL，就没有交易能被授权了。撤销也是很可靠的，能撤销多少证书并没有一个直接的上界。

中心CRL数据库也存在一些主要的缺陷，每个人必须一直在线才能够检查CRL数据库。CRL数据库还有一个很大的缺点：如果它不能正常工作，就什么事情都不能完成了。如果当CRL不能工作时，你试图通过授权证书有效来解决这个问题，那么攻击者就会使用拒绝服务攻击让CRL数据库不工作，进而破坏系统的撤销能力。

一种选择是采用分布式CRL数据库。可以用分布在世界各地的很多服务器来建立冗余的镜像数据库，并且希望它非常可靠。但是这样的冗余数据库的建立和维护代价都非常大，因此一般情况下不予采用。不要忘记，所有人都不想在安全方面投入太多。

一些系统只是把整个CRL数据库的副本送到系统的各个设备上。美国军用的STU-III加密电话就是以这种方式工作的，这就类似于把被偷的信用卡号码手册送给每个商人，这是很容易做到的。可以让每个设备每隔半小时左右从Web服务器上下载更新的CRL，这是以增加撤销的时间为代价的。然而，这种办法限制了CRL数据库的大小。你不可能花大量的时间把成百上千条CRL数据复制到系统的各个设备上。有些系统要求每个设备必须能存储50条CRL数据，攻破这样的系统是不会得到嘉奖的。

以我们的经验，CRL系统的应用和维护都是很昂贵的。它们要求有自己的基础构架、管理程序、通信路径等。为了得到很少用到的撤销功能，需要做大量的额外工作。

20.8.2 短的有效期

可以使用短的有效期来代替撤销列表，这可利用已经存在的有效期机制来完成。CA只发行很短有效期的证书，时间范围是从10分钟到24小时。每当Alice想用她的证书时，她会从CA处得到一个新的。只要它是有效的，Alice就可以一直使用。确切的有效期间隔能够根据应用的需求进行调整，但证书的有效期若少于10分钟则太不实际了。

这个方案的主要优点在于，它使用的是已经可行的证书发行机制，不需要独立的CRL，这大大降低了整个系统的复杂性。要撤销证书，需要做的仅仅是告诉CA新的访问许可规则。当然，每个人仍然要一直在线才能得到重新发行的证书。

简单化是我们的主要设计需求之一，因此与 CRL 数据库相比，我们更倾向于使用短的有效期。短的有效期方案是否可行主要取决于应用是否要求快速的撤销，或者延迟是否可被接受。

20.8.3　撤销是必需的

因为撤销很难实现，很多人倾向于根本不用它。一些 PKI 的建议中没有提到撤销，另一些则把 CRL 列为将来的可能扩展。实际上，没有撤销功能的 PKI 是没什么用的。在现实环境中密钥肯定会被泄露，访问许可权必须要被撤销。在没有撤销系统的 PKI 上操作就像行驶在没有抽水机的轮船上一样。理论上说，轮船应该是防水的，它不需要抽水机。但实际上，在轮船的底部总会有一些水，如果不清除它，轮船最终会沉下去的。

20.9　PKI 的优势是什么

在最初讨论 PKI 时我们就说明了使用 PKI 的目的，即让 Alice 和 Bob 能产生共享密钥，用它来建立安全的信道，以便能彼此安全地通信。Alice 想不通过第三方来认证 Bob（反之亦然）。PKI 被认为可以实现这一目的。

但实际上它做不到。

没有能离线工作的撤销系统。原因很显然：如果 Alice 和 Bob 都不与外界联系，他们就不能知道其中某个密钥被撤销了，因此撤销检查迫使他们是在线的。我们的两个撤销方案都要求是在线连接。

但是如果我们是在线的，就不需要这样复杂的大 PKI。我们可以建立一个中心密钥服务器来获得安全性，就像第 18 章中阐述的一样。

让我们来比较一下 PKI 与密钥服务器系统的优势。

- 密钥服务器要求每个人实时在线。如果不能连接到密钥服务器，就什么事都做不了，Alice 和 Bob 就没有办法彼此通信。而 PKI 具有一些优势。如果使用过期的撤销请求，那你需要花一点儿时间同中心服务器联系；对于使用有几个小时有效期的证书来说，在线条件就放宽了许多。即便使用 CRL 数据库，在 CRL 数据库不能连接时也会有规则说明怎样继续操作。信用卡系统就有类似的规则，如果不能得到自动的认证，任何达到一定量的交易就被认可了。这些规则建立在风险分析的基础上，包括 CRL 系统中拒绝服务攻击的风险，但至少你可以选择继续进行，而密钥服务器方案就不能提供这种选择。
- 密钥服务器是一个攻击点。分布密钥服务器比较困难，因为它包含了系统的所有密钥。你一定不想把你的密钥传遍世界。反过来，由于 CRL 数据库少了一些安全要求，因此更容易分布。短的有效期方案使 CA 成为一个攻击点，但大型系统总是有分层的 CA，这意味着 CA 是分布式的，故障只会影响系统的一小部分。

- 理论上讲，PKI 提供了不可否认性。一旦 Alice 用她的密钥签署一个消息，她以后就不能否认自己签过这个消息了。密钥服务器就不能提供这种功能。中心服务器能得到Alice的密钥，因此可以伪造任何消息，使它看起来像是 Alice 发出的。在实际生活中，不可否认性是不可行的，因为人们不可能非常好地保存密钥。如果 Alice 想要否认自己签了一个消息，她就可以说病毒“感染”了机器并偷走了她的私钥。
- PKI 最重要的密钥是 CA 根密钥。这个密钥不必存储于在线的计算机上，它可以被安全地存放在离线的机器上。根密钥仅仅用于签署子 CA 的证书，这很少用到。相反，密钥服务器把主要的密钥都放在在线的计算机上。离线的计算机比在线计算机更难被攻击，因此这使得 PKI 相对来说更安全一些。

综上所述，PKI存在一些优势。使用PKI是很好的，但不用它也会有一定的好处。因为PKI的代价太大。PKI 比密钥服务器更复杂，公钥的计算需要非常大的计算量。

20.10 选择

那么怎样来建立密钥管理系统呢？用密钥服务器类型的方案还是PKI类型的方案呢？这总是要根据具体的需求来决定。

对小型系统来说，PKI的额外复杂性是我们不希望的。我们认为用密钥服务器方案更容易一些，这主要是因为 PKI 相对于密钥服务器的优势主要体现在大的系统中。

对大型系统来说，PKI的额外复杂性仍然是很有吸引力的。PKI是一个更具分布性的系统。信用模式的扩展允许中心CA限制子CA的权限，这就使得覆盖特定区域的子CA更容易建立。由于子CA被限制在用它自己的密钥证书发行的证书中，它不能给整个系统带来风险。对于大的 CA 系统，这样的灵活性和风险控制是很重要的。

如果你正在建立一个大系统，我们建议你要重点考察PKI方案，但也要同密钥服务器方案做比较。你必须要看看PKI的优势是否超过了它的额外花费和复杂性。这里有一个问题需要说明，你可能想用信用模式的限制来处理子 CA。为此，你必须要在逻辑框架中表示这种限制。由于没有通用的框架，因此这最终是一个根据特定客户进行的设计。这似乎意味着你不能使用 PKI 中脱离框架的产品，因为它们大都没有相应的证书限制语言。

第21章 PKI的实用性

如果你需要PKI，就要决定是买一个还是构建一个。你可以购买PKI产品，这样的话你就不需要这本书了。这是一本工程学方面的书，不是用户指南。因此，我们假定你正在自己构建PKI。我们要讨论在设计PKI系统时的一些实际考虑。

21.1 证书模式

无论如何，你都不要考虑X.509证书。如果想知道原因，看看参考文献[40]就知道了。你自己来做有可能更容易。证书仅仅是带有多个域和选择域的数据类型。这是一个直接的软件工程问题，我们不详细讨论了。重要的是要注意特定数据结构的编码是惟一的，因为在密码学中我们常常对数据取Hash值来进行签名或做比较。对一个类似XML的允许同一数据结构有几种表示的模式，需要更加注意保证签名和Hash函数正常工作。

21.1.1 许可语言

除了最小的PKI系统之外，你肯定想要限制子CA发行的证书。为此，需要把权限范围编码到子CA的证书中，这就要求有一种语言来表示密钥的许可权。这在PKI的设计中可能是最困难的，我们都帮不了你。你做的限制要根据具体应用来决定。如果不能找到明智的限制方案，就应该重新想想是否应该使用PKI了。如果在证书中没有限制，那么每个子CA都会有一个主密钥——那是很不安全的设计。也可以把系统限制为一个CA，但那样就会失去很多PKI与密钥服务器系统相比的优势。

21.1.2 根密钥

无论做什么事，CA都必须有公/私钥对。生成这个对是很直接的。公钥以及有效期之类的一些其他数据被分配给每个用户。为了简化系统，通常使用一个自证实的证书，它是一种很奇怪的构造。CA用自己的公钥签署一个证书，虽然它叫做自证实的证书，但它并不能证实什么。自证实的名字是我们一直使用的历史上的误称。这个证书根本不能证实密钥，也不能证实密钥的任何安全特性，因为任何人都可以生成一个公钥，然后自己证实它。自证实的证书就是把公钥和附加数据绑定在一起。许可权列表、有效期、联系信息等都包含在自证实的证书中。它使用同系统中其他证书一样的数据格式，所有的用户都可以用现存的代码来检查这些附加数据。自证实的证书被称为PKI的根证书。

下一步要做的是把根证书以安全的方式分配给所有的系统用户。每个人必须知道根证书，而且必须有正确的根证书。

计算机第一次加入PKI时，必须以安全的方式得到根证书。这是很容易做到的，类似于给计算机指定一个本地文件或可信任的网络服务器上的文件，然后告诉机器这就是PKI的根证书。密码学对根证书的初始分布并没有帮助，因为没有用来提供认证的密钥。如果CA的私钥被泄露了，同样的情形也会发生。一旦根密钥不再安全了，整个的PKI结构将不得不初始化，这也包括以安全的方式给每个用户分配根证书。这就说明了保证根密钥安全的重要性。

根密钥在一段时间后会到期，中心CA就要发行新的密钥。发布新的根证书更容易。新的根证书是用旧的根密钥签署的，用户可以从不安全的来源处下载新的根证书。由于它是用旧的根密钥签署的，因此不能被修改。惟一可能存在的问题是用户没有得到新的根证书。大多数系统会把新旧根密钥的有效期重叠几个月，以留出充分的时间转到新的根密钥。

这里有一个小的应用问题。新的CA根证书可能有两个签名——一个是用旧的根密钥签的，使得用户能识别新的根证书，一个（自证实）是用旧密钥到期后产生的新密钥签的。你可以通过在你的证书模式中包含对多签名的支持来实施，或者仅仅是对同一个新的根密钥发行两个独立的证书。

21.2 密钥的生命周期

让我们来看看单个密钥的生命周期。它可以是CA的根密钥，也可以是任何其他的公钥。密钥在它的周期中包含几个阶段。并不是所有的密钥都需要这些阶段，这要由具体的应用来决定。作为例子，我们来讨论Alice的公钥。

产生 密钥周期的第一步是产生密钥。Alice产生一个公/私钥对，并以安全的方式存储私钥部分。

证实 下一步是验证。Alice把她的公钥传给CA或子CA，让它来证实公钥。就是在这里，CA决定给Alice的公钥赋予哪些许可权。

分发 根据具体的应用，Alice也许要在使用公钥前分发她的被证实了的公钥。例如，如果Alice用公钥来签名，能收到Alice签名的每一方都应该先有她的公钥。最好的办法就是在Alice第一次使用公钥前的一段时间内分发密钥，这对于新的根证书是相当重要的。例如，当CA转换成一个新的根密钥时，在提交用新密钥签署的证书之前，每个人都应该有机会了解新的根证书。

是否需要一个独立的分发过程要根据具体的应用来决定。如果可能，应该尽量避免这一过程，否则就要向用户解释该过程，并且要在用户界面上可见。这样的话，就会带来很多额外的工作，因为很多用户都不能理解它的方式，从而不能正确地使用系统。

主动使用 下一阶段是Alice主动用她的公钥进行交易。这是密钥使用的一般情形。

被动使用　主动使用阶段过后，一定会有一段时间 Alice 不再用她的密钥进行新的交易，但所有人仍然接受这个密钥。交易不是瞬时完成的，有时会有延迟。一个签名的电子邮件可能要用一天或两天才能到达目的地。Alice 应该停止主动地使用密钥，在密钥到期之前留出一段合理的时间，让所有延迟的交易完成。

有效期　最终，密钥到期，它不再被认为是有效的。

密钥阶段是怎样定义的呢？最通用的方案是在证书中包含每个阶段过渡的清晰时间。证书包含分发阶段的开始、主动使用阶段的开始、被动使用阶段的开始以及截止期。遗憾的是，所有这些时间不得不提供给用户，因为它们影响着证书的工作方式，对普通用户来说这可能太复杂了。

一个更灵活的方案是建立包含每个密钥阶段的中心数据库，但这会产生一系列的安全问题，我们宁愿不这样做。如果你有 CRL，就能超越被选择的阶段，很快让密钥到期。

如果 Alice 想要在几个不同的 PKI 中使用相同的密钥，情况就变得更复杂了。通常，我们认为这是一个坏主意，但有时它是不可避免的。假定 Alice 使用一个小的抗干扰组件，这个组件包含她的私钥并且做数字签名所需的计算。这样的组件具有有限的存储能力，Alice 关于公钥的证书能被存储在没有容量限制的公司内部网上，但是这个小组件不能存储无限数目的私钥。在这种情况下，Alice 最终对多个 PKI 使用同样的私钥。这也意味着在 Alice 使用的所有 PKI 中，密钥生命周期的时间表应该是类似的。做这种协调也许很困难。

如果你使用这样的系统，一定要保证在一个 PKI 中使用的签名不能在另一个 PKI 中使用。你应该总是用一个数字签名方案，正如 13.7 节所讲的那样。签名的字节串在两个不同的 PKI 系统中或在两个不同的应用中不应该是相同的，最简单的办法是在签名的字符串中包含惟一标识应用和 PKI 的数据。

21.3　密钥作废的原因

我们已经几次提到密钥会有规律地被更换，但这是为什么呢？

在理想世界中，一个密钥可以使用很长时间。不知道系统缺陷的攻击者只能进行穷搜索。理论上来说，这就把问题归结为选择一个足够大的密钥。

现实世界并不是完美的，总是存在着密钥被泄露的威胁。密钥一定会存放在某处，攻击者也许会试图得到它。密钥一定会被使用，任何使用都会带来威胁。密钥一定会从存储地点被传输到进行相关计算的地点。这常常是在单个设备内完成的，但也带来了新的攻击风险。如果攻击者能在传输的信道上窃听，那么他就得到了密钥的副本，用这个密钥就可进行密码学的操作。有安全性证明的密码函数也是没用的，这些安全性证明都是基于特定观点的，类似于“我们都没有找到攻击这个函数的办法，因此它看起来相当安全。”[1]

---

[1] 通常所说的密码函数的“安全性证明”并不是这样的。这些证明仅仅是一种归约：如果你攻破函数 A，你就能攻破函数 B。这使你减少了假定是安全的函数的数目。在这方面，证明是有价值的，但这些函数并不能提供真实世界的安全性证明。

密钥的寿命越长，使用它的次数越多，攻击者设法得到密钥的概率就越大。如果要限制攻击者得到密钥，就不得不限制密钥的使用寿命。实际上，就是让密钥作废。

还有另一个原因要限制密钥的使用寿命。假定一些不幸的事情发生了，攻击者得到了密钥，从而破坏了系统的安全，引起了某些危害。(如果你发现攻击者有了密钥，使其作废是惟一有效的办法。聪明的攻击者会尽量不让你检测到。)这种危害一直持续到这个密钥被新的密钥取代。通过限制密钥的寿命，我们也就避免了系统的继续暴露。

可见，短的密钥周期有两个优势：它们减少了攻击者得到密钥的机会，限制了攻击成功带来的危害。

那么一个合理的生命周期是多少呢？这要由具体情况来决定。更换密钥需要一定的代价，因此不可能经常更换。从另一方面来说，如果十年才更换一次密钥，可以肯定的是，转换成新密钥的函数到那时一定不能正常工作了。作为一个通用的规则，很少被使用或检测的函数或进程一般不能正常运行[①]。保持密钥长期不变的最大危险也许是更换密钥函数从来没被用过，因此当需要它时就不能正常运行了。一年的密钥生命周期可能是一个合理的最大值。

密钥更换需要用户的参与，而且代价较大，因此不可能经常更换。合理的密钥生命周期是一个月以上。周期更短的密钥应该被自动地管理。

21.4 根据应用采取行动

在不知道具体应用和工作环境的详细情况时，我们不能给出具体的答案。对于加密函数，甚至是密钥协商协议，我们都能给出详细的建议，但密钥管理却不是这样的问题，它是直接面对真实世界的。我们已经给出了很多整体的指导，我们知道这些建议对于设计密钥管理系统来说是远远不够的。但我们认为，不知道你所面临的特定情况就给出准确的建议是不可能的，一本书并不是能给出个性化建议的有效手段。

① 这是一个公认的真理，也是应该经常测试诸如消防训练紧急启动进程的主要原因。

第22章 存储秘密

我们在9.3节已经讨论了如何存储会话密钥之类的短暂秘密的问题，但是怎样存储像密码口令和私钥这样的长期秘密呢？我们有两个相反的要求。首先，秘密应该保持其私密性；其次，丢失秘密的风险（也就是再也找不到这个秘密）应该是最小的。

22.1 磁盘

存储秘密的一个很直接的办法是把秘密存在计算机的硬盘上或其他的长期存储媒介上，这只有在计算机是安全的情况下才能工作。如果Alice在她的PC机上存储她的密钥（没有加密），那么任何使用其PC的人都能使用她的密钥。大多数PC机都能让其他人使用，至少偶尔是这样。Alice不会介意让其他人用她的PC，但她肯定不想让他人同时进入自己的银行账户。另一个问题是，Alice可能使用几个计算机。如果她的密钥存储在家里的PC上，那她工作时或旅行时就不能使用它。那么她应该把密钥存在家里的台式电脑里还是她的膝上电脑里呢？我们真是不想让她把密钥复制到多个地方，那样会减弱系统的安全性。

一个更好的解决办法是让Alice把密钥存储在她的PDA上。PDA很少会借给他人使用，Alice无论去哪里都会随身带着它。（另一个选择是用Alice的手机或手表，但是需要更新它们的底层结构，这超出了我们的范围。）

你可能会想到，如果我们对秘密加密，安全性将有所改进。确实是这样，但用什么来加密呢？我们需要用主密钥来加密，而主密钥需要被存储在某处。把它存成被加密的秘密不会给你带来任何优势。但这是减少秘密信息的大小和数目的有效技术，它与其他技术合起来被广泛使用。例如，RSA的私钥有几千比特长，但通过用对称密钥对它加密，可以减少安全存储的空间。

22.2 人脑记忆

另一个办法是把密钥存储在Alice的大脑中。我们让她记住一个密码口令，用这个密码口令加密所有其他密钥。被加密的密钥可以存储在任何地方——也许在磁盘上，但它也能存储在Web服务器上，Alice能把它下载到她正在使用的任何机器上。

人类记忆密码口令的能力是相当差的。如果选择很简单的密码口令，又不能得到任何安全保障。实际上，根本不存在足够简单的又是相当安全的密码口令：攻击者仅仅通过试遍所有的可能就可破获。用你妈妈少女时的名字也不安全，她的名字通常是公开的——即便没有公开，可能只存在几十万个姓氏，攻击者可以通过尝试从而找到正确的那一个。

好的密码口令应该是不可预测的。换句话说，它必须包含很多熵。像password（密码口令）这样的一般单词包含的熵不多。据统计，有大约50万英语单词——这是在没有删节的包含了所有很长的和模糊的单词的词典中统计出来的——因此用一个单词作为密码口令最多只提供19比特的熵。在英文课本中稍微有些不同，熵的评估大约是每个字母1.5～2比特。

在整个系统中我们要用256比特的密钥来获得128比特的安全水平。在大多数情况下，使用256比特的密钥几乎不需要额外的花费。然而，在用户记忆密码口令（或密钥）的情况下，长的密钥带来的额外花销是很大的。试图使用具有256比特的熵的密码口令负担太重了，因此，我们把密码口令限制到仅有128比特的熵①。

如果我们乐观地估计每个字符有2比特的熵，那么我们需要64个字符来得到128比特的熵。这是不能接受的，用户会拒绝使用这么长的密码口令。

如果我们接受64比特的安全水平，那又会怎样呢？以每个字符有2比特的熵来计算，我们至少需要32个字符长的密码口令。即便这样，对用户来说也是太长了，很难处理。不要忘记，大多数实际的密码口令只有6~8个字符长。

你会试图使用一个被指定的密码口令，但如果系统告诉你密码口令是“719327582742-9946905186”或“aoekjk3ncmakwe”，你会使用它吗？人们没有能力记住这样的密码，因此这种办法是不可行的。（实际上用户会把密码口令写下来，这个问题将在下一节讨论。）

一个更好的解决办法是使用密码短语，它与密码口令类似。事实上，它们非常相似，我们可以认为两者是等价的。它们的不同仅仅是：密码短语比密码口令更长一些。

也许Alice会用这样的密码短语“Pink curtains meander across the ocean”（粉红的帘子蜿蜒穿过海洋）。这句话没什么意义，但很容易记住。它有38个字符长，因此可能包含大约57~76比特的熵。如果Alice把它扩展成“Pink dotty curtains meander over seas of Xmas wishes”（粉红色带点的帘子蜿蜒穿过圣诞节的希望之海），她就会得到52个字符，是一个有78~104比特熵的合理的密钥。Alice能在几秒内敲出这个密码短语，比敲一串随机数要快很多。密码短语比随机数据更容易记忆，因此可把随机的数据转换成类似的密码短语，很多记忆法则都是建立在这种想法的基础上的。

有些用户不喜欢在键盘上敲太多的字符，因此他们选择的密码短语稍微有些不同。“Wtnitmtstsaaoof, ottaaasot, aboet。”怎么样？这看起来就像是在胡说。但如果你把它看成是一句话中每个单词的第一个字符，就会明白了。我们用的是莎士比亚的一句名言：“Whether'tis nobler in the mind to suffer the slings and arrows of outrageous fortune, or to take arms against a sea of troubles, and by opposing end them。”当然，Alice不应该使用文献中的句子，因为攻击者也可拿到文献，Alice书架上的书里又有多少合适的句子呢？相反，她应该用她自己造的句子，即其他人不会想到的句子。

---

① 对于数学家来说，是从128比特熵的概率分布中选择口令。

与完整的密码短语相比，每个单词取第一个字符的技术需要用更长的句子，但它的安全性高，因为这种方法更随机。我们并不知道这种技术中每个字符的熵应该如何来评估，也许某些人会做一些研究并写出关于如何选择密码短语的文章来。

密码短语无疑是在人脑中存储秘密的最好方法。遗憾的是，很多用户对正确地使用该方法仍然感到很困难。即便是用密码短语，在人脑里储存128比特的熵也是相当难的。

22.2.1 加盐和扩展

为了从有限熵的密码口令或密码短语中产生最大的安全性，我们可以使用两种技术。这些技术是如此的简单和明显，以至于可以应用在所有密码系统中。

首先是加盐（salt）。盐是一个随机数，与加密的数据存放在一起。如果可以的话，就采用256比特的盐。

下一步是扩展（stretch）密码口令。扩展一定是很长的计算。令p是密码口令，s是盐。使用任意密码强（cryptographically strong）的Hash函数h，计算：

$$
\begin{aligned}
x_0 &:= 0 \\
x_i &:= h(x_{i-1} \| p \| s) \quad \text{其中 } i = 1, \ldots, r \\
K &:= x_r
\end{aligned}
$$

用K作为密钥来实际加密数据。参数r是计算重复的次数，它应该由实际情况来决定。（这里没有说x_i和K应该是256比特长。）

让我们从攻击者的角度看看这个问题。给定盐s和用K加密的数据，你试图尝试用不同的密码口令来找到K。选择一个特定的密码口令p，计算相应的K，用它解密数据来检查它是否为真。如果不是，则p必定是假的。为了检查单个的p，必须要做r次不同的Hash计算。r越大，攻击者做的工作就越多。

如果是正常使用，每次用到密码口令时都要进行扩展计算。但是要记住，扩展计算要在用户刚键入密码口令时完成。键入密码口令可能要用几秒钟，因此用200 ms来进行密码口令处理是可以接受的。这里是我们选择r的规则：选择r使得在用户设备上由(s, p)计算K用200～1000 ms。计算机的速度在变快，因此r也应随着不断增大。理想地说，当用户第一次设置密码口令时要通过做实验来决定r，然后把r和s存放在一起（要保证r是一个合理的值，不太大也不太小）。

我们能得到多少呢？如果$r = 2^{20}$（刚超过100万），那么攻击者不得不对他尝试的每个密码口令做2^{20}次Hash计算。尝试2^{60}个口令，要做2^{80}次Hash计算，因此有效地使用$r = 2^{20}$能让口令的有效大小增大20比特。选择的r越大，获得的口令就越大。

以另一种方式来看这个问题。r所做的是阻止攻击者从越来越快的计算机中受益，因为计算机越快，r也就越大。这是一种摩尔定律的补偿。十年后，攻击者可以用这十年发展起来的

技术来攻击你今天使用的口令，因此你仍然需要多留出一些安全额度，在口令中尽量给出更多的熵。

这也是使用有前向安全性（forward secrecy）密钥协商协议的另一个原因。无论什么应用，Alice的私钥似乎最终都用一个口令来保护。十年后，攻击者能够找到Alice的口令，得到其私钥。但是如果被加密的密钥仅仅用于运行前向安全的密钥协商协议，那么攻击者将找不到任何有价值的东西。Alice的密钥不再是有效的（它已经过期了），知道她过去的私钥也不能揭示十年前用的会话密钥。

当攻击者同时攻击大量密码口令时，盐能够防止他利用规模扩大而节约计算量。假定系统中有100万个用户，每个用户存储一个包含他的密钥的加密文件，每个文件是由用户的扩展密码口令来加密的。如果我们不使用盐，那么攻击者就可以进行如下的攻击：猜测密码口令p，计算扩展密钥K，试图用K解密每个密钥文件。对于每个密码口令来说，扩展函数仅仅需要计算一次，最终的扩展密钥可以被用于尝试解密每个文件。

当我们把盐加入到扩展函数中时，就不可能再这样做。所有的盐都是随机值，因此每个用户会使用不同的盐值。现在攻击者将不得不对每个文件计算一次扩展函数，而不是对每个口令计算一次。这对于攻击者来说计算量很大，而用户只是加入了盐值。

那么需要多大的盐值呢？我们不能做出详细的分析。也许可以不用128比特（如果不可能出现生日攻击），但为什么不用呢？比特价值低廉，用256比特的盐对每个人来说都很简单。

顺便说一下，当你做这些时一定要小心。我们以前看到过这样一个系统，该系统几乎在所有方面做得都很好，但某个程序员想要对用户输入的口令是否正确给用户一个快速的答复，以此来改进用户界面，因此他在口令上存放了一个校验数，这就完全破坏了整个的加盐和扩展过程。如果反应时间太慢，可以减小一点r，但要保证在没达到r次Hash计算时不能识别口令的正确与否。

22.3 便携式存储

存储秘密的另一个办法是在计算机之外存储。最简单的存储形式是把密码口令写在一页纸上。大多数人使用这种办法，甚至对于像网站这样的非密码系统也如此。很多用户至少有半打的密码口令要记住——我们肯定有——但那是太多了，因此为了记住口令，用户就把它们写下来。这种办法的局限性在于，每次使用口令时仍然要通过用户的眼睛、大脑和手指来控制。为了把用户的错误限制在一个合理的界限内，这种技术仅仅使用熵比较低的密码口令和口令短语。

作为设计者，不必为这种方法进行设计和应用。无论怎样制定规则，怎样产生密码口令系统，用户都会这样来存储口令。

一个更先进的存储方法是便携式存储。它可以是一个存储块、一张软盘、一个条纹卡或是其他的数字存储设备。数字存储系统总是相当大，能存储256比特的密钥，因此我们可以不采用低熵的口令。便携式存储器非常像一个密钥，无论谁得到它，都能够进入，因此需要安全地放置。

22.4 安全token

一个更好的也是更昂贵的解决办法是使用我们称为安全token的设备。这是Alice能够带在身边的一个小型计算机。token的外部形状很不一样，从智能卡（看起来就像信用卡）到iButton、USB dongle或PCMCIA卡。它的主要特性是有一个非易散型的内存（例如，没电时仍能保存数据的内存）和一个CPU。

安全token主要作为一个便携式存储设备来工作，但它在安全性上又有几点改进。首先，通过口令或类似的指令来限制访问被存储的秘密。在安全token让你使用密钥之前，一定要给它一个正确的口令。如果输入了三五次错误的密码口令，安全token就不允许用户进入了，通过这种方式可以防止攻击者对密码口令进行穷举搜索。当然，某些用户经常错误地输入密码口令，那么他们的安全token就应被恢复，但要用更长、熵值更高的密码短语或密钥来保证安全性。

这种做法可以提供多层保护。首先，Alice在物理层面上保护了token，例如，可以把它放在钱包里或钥匙链上。攻击者不得不去偷这个token，或者以某种方式进入到token里。那么攻击者或者需要以物理方式打开这个token抽取数据，或者通过找到密码口令来打开token。token常常是抗干扰的，这使得物理攻击更困难①。

安全token目前是最好的也是最实际的存储密钥的方法之一。它们相对来说不是很昂贵，并且很小，携带很方便。

实际使用中要注意的一个问题就是用户的行为。当他们去吃饭或开会时，他们会把插在计算机中的安全token留在那里。由于用户不想每隔很短的时间间隔就需要输入密码，因此系统被设置成在密码最后一次被键入的几小时内是允许访问的，所以攻击者就可以走过去，开始使用存储在安全token中的密钥了。

你可以通过培训来解决这个问题。目前有很多关于"办公室的安全性"的视频介绍，还有"带着您的安全token去午餐"的海报，以及"如果我再发现你的安全token插在那儿无人管，你就还会得到这样的训话"之类的演说。但你也可以用其他的方式。让安全token不仅是访问数据的密钥，而且也是办公室的门锁，这样用户就不得不带着安全token回到办公室。把咖啡机配置成仅仅当提交安全token后才能输出咖啡，这就促使雇员在喝咖啡时把安全token带到咖啡机那儿，而不是把安全token留在计算机上。安全性有时的确包含这样的措施，这些措施比强迫"把安全token带在身边"的制度实施起来效果更好。

22.5 安全UI

安全token仍然存在严重的缺陷。Alice使用的口令不得不输入到PC或其他的设备中。只要我们信任PC，这就不是问题，但我们所知道的PC都不是很安全的。事实上，不把Alice的

① 它们是抗干扰的，但并不能防止干扰。干扰总是会发生的，抗干扰仅仅是使干扰更不容易发生。

密钥存在PC中的主要原因就是不太信任它。如果token本身有一个安全的内置UI，我们就能得到更多的安全性。考虑一个有内置键盘的安全token，现在，口令或PIN就能够直接输入token而不需要信任外部的设备了。

把键盘嵌在token中能防止PIN被攻破。一旦PIN被敲击，PC仍然会得到密钥，那么它就能用这个密钥做任何事情，因此我们仍然被整个PC的安全性所限制。

为了阻止密钥的泄露，我们必须把包含密钥的密码进程嵌入token，这就要求在token中有专用的代码。这个token很快就成为一台功能很全的计算机，并且是一个用户能随身携带的可信任的计算机。这个可信任的计算机能把每个应用的主要安全部分在token中实现。显示功能现在变得很关键，因为通过敲击用户的PIN，显示功能会告诉用户他正在授权什么行为。在一个典型的设计中，用户使用PC的键盘和鼠标来进行操作。例如，当银行支付被授权时，PC把数据发送给token。token显示数量和其他一些交易细节，用户通过敲击他的PIN来授权交易，然后token签署交易细节，PC完成交易的其余部分。

在现实中，带有安全UI的token对大多数应用来说太昂贵了。也许功能最相近的产品是个人数字助理（PDA），PDA像一种流行的掌上电脑平台Palm。然而，人们把程序下载到PDA上，PDA从开始就没被设计成一个安全单元，因此它可能还没有PC机安全。

这是一个安全性与功能性相抵触的好例子。人们愿意下载程序，并且希望能随时运行它。他们也愿意相信计算机能提供有价值的信息。以现有操作系统的安全状态来说，在一个单个计算机中既有下载程序的灵活性、又有我们所需的安全性是不可能的。通过在屏幕上显示跳舞的小猪可以让用户在下载程序和安全性之间进行选择。

22.6 生物统计学

我们还可以把生物统计学的知识结合进来。你可以把类似指纹扫描仪或虹膜扫描仪的设备嵌入到安全token中。但现在，生物设备并不是很有用。指纹扫描仪生产成本合理，但所提供的安全性却是无法估测的。2002年，密码学家Tsutomu Matsumoto和他的3个学生演示了能买到的所有类型的扫描仪都是怎样被欺骗的，而且仅仅是采用很普通的工具[64]。即便是从潜在的指纹（例如，留在每个光面上的指纹）中伪造一个假指纹，也仅仅是一个聪明的中学生凭借业余爱好就可以完成的。

令我们真正感到的不是指纹扫描仪能够被欺骗，而是欺骗的手段是如此的简单，代价又是如此的小。更糟糕的是，生物工业一直在告诉我们生物识别是多么安全，他们从来就没说伪造指纹是这么容易。现在突然来了个数学家（甚至都不是生物学家）让整个过程浮出水面。有两种可能性：或者是生物工业界已经知道这种情况，然而却对我们撒谎；或者是他们根本不知道，因此是明显的不称职。你真要考虑考虑其他的生物系统的安全性了。

虽然指纹扫描仪很容易伪造，但仍然是很有用的。假定你有一个安全token，它带有一个小显示器、一个小键盘和一个指纹扫描仪。为了得到密钥，你需要对token进行物理控制，得

到PIN，伪造指纹。与以前的方法相比，攻击者需要做更多的工作。它可能是目前我们所能达到的最实际的密钥存储方案。另一方面，安全token是相当昂贵的，因此它不能被很多人使用。

指纹扫描仪也可以被应用在安全性要求不高的方案中。用手指触摸扫描仪是很快的，让用户经常这样做也是可行的。指纹扫描仪可以用于增加只有特定的人才能授权计算机做某事的信任度。因此，即使某雇员把口令告诉了同事，而后者也不容易操作了。指纹扫描仪不能防范很高级的攻击者，它可用来防止安全制度的临时破坏。与把扫描仪用做安全性高的设备相比，这对安全问题来说也许是一个更重要的贡献。

顺便说一下，Matsumoto的工作也许对于在刑事案件中使用指纹作为证据有很大的影响。它已经证实从潜在的指纹中进行伪造是很容易的。那么怎样把一滴汗放在伪造的指纹中并用它来代替放在某处的指纹呢？到现在为止这仍然是间谍电影的情节，但它也许很容易实现。这样一来，凶器上的指纹与过去的含义不再一样了。

22.7 单点登录

一般的用户都有很多密码口令，因此产生一个单点登录系统就变得很有吸引力。办法是给Alice一个单一的主密钥，然后用它来给不同应用中的不同密码口令加密。

为此，所有的应用都必须对单点登录系统对话。无论何时，只要应用需要密码口令，都不应该询问用户，而是由单点登录程序来完成。实际上，这个系统并不能正常工作。这个过程并没有广泛应用的标准，标准产生之前，它不会自动完成。让我们想想这种情形吧：所有不同的应用都自动地改成从单点登录系统中取得密码口令。

更简单的办法是用一个小程序把密钥都存在一个文本文件中。Alice敲击她的主密钥，然后用复制和粘贴功能把密钥从单点登录程序复制到应用中。Bruce设计了一个叫做安全密钥的公开主程序来完成这一操作，但这仅仅是Alice把她的密钥写在一张纸上的数字版本。如果你总是使用同一台计算机，则对于上述改进是有益的，但单点登录的办法肯定不是最终的解决方案。

22.8 丢失的风险

但是，如果安全token被攻破会怎么样呢？或者是把带有密码口令的纸忘在兜里，让洗衣机给洗了呢？丢失密钥总是很糟糕的事情。丢失的代价是至少不得不给每个应用注册一个新密钥，严重的话会永远失去访问重要数据的许可权。如果你用密钥加密的是你花了五年时间完成的博士论文，而你把密钥丢失了，那么你就再也不能找到这个论文了，你能看到的只是含有一堆随机字节的文件。太令人心痛了！

让一个密钥存储系统既容易使用又高度可靠是很难的。因此，把这些功能分开是一个有效的规则。保持密钥的两个副本——一个容易使用，另一个更可靠。如果容易使用的系统忘记了密钥，可以从可靠存储的系统中恢复它。找到可靠的系统很简单，写在银行地下室的一页纸上怎么样呢?

当然，你一定要小心保护你的可靠存储系统。通过设计，它很快会用于存储所有密钥，那么这就很容易成为攻击者的目标。你应该做一个风险分析以决定是用许多小的可靠系统存储密钥好，还是用一个大的更好一些。

22.9 秘密分享

你需要更安全地来存储某些密钥——例如，CA中私有的根密钥。我们已经看到，以安全的方式存储秘密是很困难的，安全可靠地存储它就更困难了。

有一种密码学的方法能够帮助我们存储密钥，它叫做秘密分享（secret sharing），这有些用词不当，因为它暗示着你要同几个人分享秘密。实际上不是这样的，该方法是把这个秘密分成几个不同的分享（share）。它可能以这样的方式来处理，例如，用5个分享中的3个就可以恢复秘密。那么你可以把分享分给IT部门的每个主要成员，他们中的任何3个人都能恢复秘密。这种方式的诀窍就是任意两个人在一起绝对不会知道关于密钥的任何事情。

从学术观点来说，秘密分享系统是很有吸引力的。每个分享用我们以前讨论的一种技术来储存。一个从n中抽出k个的规则把高度安全性（至少有k个人才能获得密钥）与高度可靠性（$n-k$个分享丢失也不会有损害）合并在一起。有一种更奇特的秘密分享方案，它允许更复杂的访问规则，以具体的（Alice和Bob）组合或（Alice、Carol和David）组合才能进入。

在现实生活中，秘密分享方案很少被使用，因为它们太复杂了。它们不但应用起来复杂，更重要的是，管理和操作也很复杂。大多数公司没有一群很负责的彼此信任的人。你可以尝试去告诉某个部门的成员，就说会给他们每人配备一个带有秘密分享的安全token，并且他们要在星期日早晨3点的紧急情况中露面。是的，他们彼此之间并不信任，但却要保持自己的分享安全，不被其他成员盗用。每次当某人加入或离开部门时，他们也需要去安全密钥管理房间以得到一个新的密钥分享。实际上，这意味着在部门成员出差的情况下也要这样。CEO持有分享也不是很有用的，因为CEO更容易出差。在你知道出差之前，要把它交给两个或三个高级IT管理人员。他们能够使用秘密分享方案，但代价和复杂性已经使它没有吸引力了。为什么不使用更简单的方案呢？例如，像银行地下室这样的物理解决方案。它有几个优势：每个人都理解它是怎样工作的，因此不需要特殊的培训。他们已经被测试过了，而秘密重构过程是很难测试的，因为它需要大量的用户交互行为——你一定不想在秘密重构过程中出现bug，那会导致你失去CA的根密钥。

我们不去详细地解释秘密分享方案是怎样操作的。这种解释需要一些数学背景，由于秘密分享方案很少被使用，所以在这里讲这些没什么意义。

22.10 清除秘密

任何存储的长期秘密最终都要被清除。只要秘密不再被需要，它的存储就应该被清除，以免将来被泄露。我们在9.3节已经讨论了清除存储器的问题，而从长期存储介质中清除长期秘密就更难了。

我们本章讨论的存储长期秘密的方案使用了各种数据存储技术：硬盘、纸张、软盘、磁条卡、可擦可编程只读存储器（EPROM）、电可擦除只读存储器（EEPROM）、闪存或有电池的随机存储器（RAM），这些存储技术中都没有保证存储数据不再被恢复的文件清除功能。

22.10.1 纸

要清除写在纸上的密码一般是把这张纸销毁。一种可能的方法是烧掉这张纸，然后把灰烬碾成粉末，或者是用水把灰烬混成纸浆。把纸切成碎片也是一种选择，虽然很多碎纸机切成的碎片很大，这就使得把纸重新合成起来更容易一些。

22.10.2 磁存储介质

磁介质是很难清除的。几乎很少有文献讨论怎样对其进行清除，我们所知道的最好的一篇文章是 Peter Gutmann[39]写的，虽然其中的一些技术细节现在可能已经过时了。

磁介质在小型磁区域中存储数据，区域的磁化方向决定了它编码的数据。当数据被重写时，磁化方向也被转变以反映新的数据。但是有几个装置能防止旧的数据完全丢失。用来重写旧数据的读/写头从来就没有被准确定位，它会留下一些旧的数据。重写并不能完全清除旧的数据。你可以把它想像成用单层涂料来重新刷墙，你仍然会模糊地看到下面的旧涂料。磁化区也会移至读写头控制的磁道外或深层磁化材料中，在那里数据可存放很长时间。用一般的读/写头来重写数据是不能恢复的，但拆开磁盘驱动器并使用特殊装置的攻击者也许会恢复部分或所有的旧数据。

实际上，用随机数据重复地重写秘密可能是一个最好的办法。这里要记住以下几点：

- 每次重写应该使用新的随机数据。一些研究工作者已经开发了特定的数据模式，这些模式被认为能更好地清除旧数据，但模式的选择还要由磁盘驱动器的具体细节来决定。随机数据也许会要求对同一结果进行更多次的重写，但它在所有情况下都能工作，因此更安全。
- 一定要在存储秘密的实际位置重写。如果仅仅通过给文件写入新数据来改变它，文件系统也许会决定在不同的位置存储新数据，这会保留原始数据。
- 要保证每次重写是写到磁盘上，而不是写到磁盘缓存中。这种写缓存的磁盘驱动器存在一定的危险，因为它们也许会缓存新数据，并把多个重写操作优化为一个。
- 清除从秘密数据存入之前到数据存入之后的那段区域也许是一个好办法，因为磁盘驱动器的旋转速度从来都不完全一致，新的数据并不能准确对准旧的数据。

据我们所知，没有一个可靠的信息能说明需要进行多少重写操作，但我们没有理由选择较少的次数。要清除的仅仅是单个密钥（如果你有很多秘密数据，用这个密钥加密那些数据，仅仅清除密钥即可）。我们认为用随机数据进行50次或100次重写是比较合理的。

使用消磁机来清除磁带或磁盘从理论上说是可行的。然而，现代的高密度磁盘存储介质能够抵抗消磁，因此这不是一个可靠的清除方法。实际上，用户根本无权使用消磁机，因此这也是不现实的。

甚至对于扩展了的重写都会有很专业的、得到资助的攻击者从磁介质中把秘密恢复出来。为了完全地清除数据，可能不得不清除介质本身。如果磁化层是塑料的（软盘或磁带），可以把它切碎，然后烧掉。对于硬盘，可以用磨光机把磁化层从盘中取出，或是用喷灯把磁盘熔化成液体。实际上，你并不想让用户采取这么极端的措施，因此重复地重写是最实际的解决办法。

22.10.3 固态存储

清除像EPROM、EEPROM和闪存这样的非易失性内存也会有同样的问题。对旧数据进行重写不能根本地清除旧数据，我们在9.3.4节讨论的数据保留装置也会起作用。而且，用随机数据重复地重写秘密是仅有的一种实际解决方案，但也不是很完美的方案。因此只要不再需要某种固态设备了，就应该销毁它。

第四部分

其他事宜

第23章 标准

第24章 专利

第25章 专家

致谢

参考文献

第23章 标　　准

很显然，本书至此还没有谈论标准，其原因是很少有正在使用的安全标准。但是如果你从事密码学的工程，就会遇到一些标准，因此需要知道一些。

23.1 标准过程

对于还不知道标准发展过程的人来说，我们首先要阐述一下标准是怎样产生的。最初，有一些标准化组织，像Internet工程任务组（IETF）、电气和电子工程师协会（IEEE）、国际标准化组织（ISO）及欧洲标准化委员会（CEN），这些标准化组织建立一个委员会来满足新标准或改进标准的需要。委员会有不同的名字：工作组、任务组或其他名字。有时会有委员会的分层结构，但基本的思想是一样的。委员会成员一般都是自愿参加的，人们申请加入，只要还不错都可被接收。通常申请者需要通过一些过程化的考试，但成员的选择没有很重要的界限。这些委员会最多有几百人，但大的委员会要分成几个小的子委会（叫任务群、研究群等）。大多数工作都是由几十人的委员会来完成的。

标准化委员会有固定时间的会议，每隔几个月召开一次。所有的成员都来到一个城市，在宾馆开几天会。在会议之间的几个月里，委员会成员要做一些工作，提出建议和陈述，等等。通常有一个编辑，他的工作是把所有的提议放在一个标准文件中。产生一个标准是很慢长的过程，一般要花几年的时间。

那么谁来参加这些委员会呢？成为一个成员是要付出很大代价的。即使不考虑时间问题，路费和住宿费也都不便宜，因此每个成员都是由公司派送来的。公司有几种动机：有时他们想要卖的产品必须和其他公司的产品一起使用。这就要求标准化，跟踪标准化过程的最好方式就是参加标准化委员会。公司也想盯住他们的对手。你不会想让你的对手去写标准，因为他们能做一些使你们处于劣势的事情——也许标准会朝着他们自己的技术和需求方向倾斜，或者包含他们持有专利的技术。有时公司不想要标准化，因此他们在委员会的会议中尽量让标准化进程慢一些，从而给他们的产品留出占有市场的时间。在现实生活中，所有这些动机再加上一些其他的因素，混合在一起构成了一个很复杂的政策环境。

毫无疑问，有很多标准化委员会失败了。他们或者从来没产生过任何标准，或者产生的标准相当不好，或者是最终形成僵局，被市场所超越。成功的委员会在几年后设法推出一个标准化文件。

一旦标准被写成，每个人都要遵循它，应用它。这当然会导致不能协调合作的系统，因此产生了第二个过程，即测试协同工作能力。每个制造商都要调整他们的应用，使其能在一起工作。

这个过程有很多问题。委员会的政策结构几乎没有把产生好的技术标准放在重要位置上，最重要的事情是达成一致。当每个人对结果都持一致的态度时，标准化才能完成。为了安抚不同的派别，标准有很多选择、扩展功能、没用的替换，等等。每个派别都有他们自己的想法、意见和重点，因此最好的妥协也常常是很矛盾的。很多标准内部都是不一致的，或者甚至是矛盾的。

在推行标准化的过程中，公司也正在实施建立在标准化草案上的应用，这个事实就使得整个过程更加复杂了。因为某些人已经应用了标准草案，因此不想改动，这就使得做改变更难了。当然，不同的公司以不同的方式进行应用，他们在委员会中进行斗争，让最终的标准与他们的应用相适应。有时，仅有的妥协是选择所有公司都没有应用的标准，以使他们都具有同等的不满意度。技术优势在这种类型的讨论中并没有真正发挥作用。

23.1.1 标准

结果之一就是大多数标准相当难读。标准化文件由委员会设计，在设计过程中他们没有让文件清晰、简洁、准确或可读性强的压力。事实上，一个非常不易读的文件更容易运作，因为只有几个委员会成员理解它，能应用它，从而不会被其他成员干扰。阅读几百页写得很糟的文件是相当无趣的，因此最终大多数成员都不去阅读整个草案，而是仅仅阅读他们感兴趣的部分章节。

23.1.2 功能性

正像我们已经提到的那样，协同工作能力测试总是需要的。当然，不同的公司会应用不同的标准。通常，最终所应用的与标准所定义的有一些细微的不同，这是因为每个公司已经在市场上推销它的产品了，有时标准制定得太晚了，某些东西已无法改变了。我们已经看到，商标*A*的产品认识到商标*B*的应用偏离了标准，因此就调整各自的行为来使其正常运作。

标准通常包含大量的选择，但是实际的应用仅仅使用了特定的部分，当然会有一些限制和扩展，因为标准化文件本身阐述了一些并不能正常工作的内容。当然，实际应用和标准之间的不同并没有在文件中阐述。

整体来说，整个过程仅仅强调主要的功能。无线网络允许连接，但对不同销售商的产品来说似乎并不都有管理功能。简单的HTML页在所有的浏览器上都能正确地显示，但更先进的布局特征在不同的浏览器上会给出不同的结果。我们对此已经非常习惯了，所以很难注意到。

这真是很难过的事情。作为一个行业，我们似乎不能产生一个可读或正确的标准，而仅仅是提供不同产品之间可协同工作的最基本功能。

23.1.3 安全性

这些失败意味着产生标准的一般方法并不能提供安全性。在安全方面，我们会有主动的攻击者，他们能找到离标准最远的角落。安全性也要由最弱的连接来决定：任何一个错误都可能是致命的。

我们已经强调了简单的重要性。然而标准都不是简单的。委员会标准化过程排除了简单化，产生的标准比委员会成员能够完全理解的标准还要复杂。正是基于这个原因，最终结果都不是安全的。

当我们对标准化委员会成员谈论这个问题时，常常得到类似这样的答复："是的，急躁的人总是想产生很完美的标准。"……"政策上面临的现实使我们不得不做出妥协。"……"那正是系统工作的方式。"……"看看我们已经完成的事情吧。"……"情况正在变得越来越好。"我们很难过，现在的安全性并不是足够好。标准提出后还需要进行协同性测试，这一事实就说明标准并不是很安全的。如果标准的功能部分（例如，容易的部分）并不是很完善，只有经过测试才能建立协同工作的系统，那么安全部分不可能不经过测试就达到安全性。当然，有可能产生一个安全的应用，它包含了标准的一部分功能，但这对一个安全标准来说并不是足够的。安全标准声称，如果你坚持遵守它就能达到一定的安全水平。安全是非常重要的，以至于不能把它交托给委员会。

我们不知道有哪个委员会写出的安全标准是有价值的，因此无论何时，只要某人建议使用密码标准，我们总是非常不情愿——有时甚至表现得很不友好。

这个领域有几个有用的标准，但没有一个是委员会写出的。有时仅仅是一小群人研发的一个连贯设计。有时是结果作为标准被采用，而没有很多政策上的妥协。这些标准是很好的，我们下面来讨论最重要的两个。

23.2 SSL

SSL是Web浏览器用来同Web服务器进行安全连接的安全协议。第一个被广泛使用的版本是SSL 2，它包含了几个安全漏洞。改进后的版本是SSL 3[37]。它是由三个人设计的，没有经过任何委员会的过程。SSL 3已经被广泛使用，并被普遍认为是一个好协议。

警告：SSL是一个好协议，但这并不意味着任何使用SSL的系统都是安全的。SSL依靠PKI来认证服务器，对嵌入在大多数浏览器中的客户进行访问是非常容易的，因此整体的安全水平相当低。其中一个浏览器有来自35个不同CA的108个不同的根证书，因此在考虑主动攻击之前，我们必须信任遍及世界的35个不同组织和Web信息。

SSL从来就没有被真正地标准化，它仅仅被Netscape应用，变成了一个事实上的标准。SSL的标准化和进一步开发正在由IETF工作组实施，叫做TLS。我们还没有详细地研究过TLS。到目前为止，它的改变非常小，我们没有理由认为TLS不如SSL 3好。但是从IETF最近设计的像IPsec[33]这样的安全协议来看，肯定存在着委员会为了显示权威而毁掉一个好标准的危险。

23.3 AES：由竞争带来的标准化

对于我们来说，AES是标准化安全系统的一个突出例子。AES不是由委员会设计的，而是由竞争得来的设计。开发过程非常简单。首先要说明系统最终达到的目标，这种说明可以由来自不同源的一小群人提出。

下一步就是收集建议。可以请专家来开发满足给定要求的完整解决方案。一旦给出建议，余下的事情就是在建议中做出选择。这是一场直接的竞争，可通过多种尺度来评价这些方案。只要把安全作为主要的衡量尺度，建议提交者就会一直关注寻找他们竞争者的建议中存在的安全漏洞。幸运的话，这会带来有用的信息反馈。在其他情况下，也许要请外边的专家来进行安全评估。

最终，如果幸运的话，你就能选择一个建议，或者不加改动，或者做一点改动。这并不是把不同的建议进行融合，那样就会导致另一种委员会设计。如果没有一个建议能够满足要求，而且似乎能产生一个更好的建议，那么就应该去寻找新的建议。

这正是NIST进行AES竞争的方式，它进行得非常好。初始的15个建议在第一轮评估后剩下5个候选。第二轮的评估最终会选出获胜者。令人惊讶的是，5个候选建议中的任何一个都是好标准，肯定比任何委员会设计的标准都要好。

如果没有足够的专家至少提出几个有竞争力的设计，标准化的竞争模型就不能工作。但我们的建议是，如果没有足够多的专家提出几个设计，就不应该对任何安全系统进行标准化。为了达到简单化和一致性，这对整个系统安全是至关重要的，安全的系统必须由一小群专家来设计。然后需要其他的专家来分析建议并对它进行攻击，寻找漏洞。为了得到好的结果——无论采用什么过程——都需要足够的专家来形成至少3个建议组。如果有很多专家，就应该使用竞争模型，因为它是到目前为止已经证实的能够产生好的安全标准的惟一模型。

第24章 专 利

专利对密码学有着更广泛的影响。我们理解专利的业务原理，很多专利都有我们的名字（通常，专利是我们所做工作的结果，被雇佣我们的公司所拥有）。但是专利常常意味着我们不能使用所需要的数学工具，强迫我们以不喜欢的方式达成妥协。更重要的是，我们不相信专利能服务于安全社会。有几方面的原因使得目前的专利系统在实践中并不能被很好地应用。

24.1 先知部分

通常假定不能把已经为众人所知的事情当成专利。我们把在专利被应用之前已经被广泛知道的事情叫做先知部分（prior art）。但是，正如使用当前专利系统的人们所知道的，这不会阻止你声明一些已经为人所知的事情。2001年，澳大利亚的一个小伙子发明了轮子的专利，或者说是一种“循环的传输简易装置”[52]。对于这个专利你会想到可能有一些先知部分，由此我们就可以看出专利系统是多么的糟糕。大多数专利在系统的滥用中并不是很值得炫耀的，但我们看到专利声明很多次地扩展并覆盖先知部分。有一个专利是某人在参加会议6个月后申请的，而他的专利技术正是在这个会议上由其他人提出的。

先知部分被专利霸占的原因是，专利办公室不能检查一个专利声明是否覆盖了先知部分。周围有太多以前的结果，你不可能把它们都找出来。问题是一旦专利被公布，就被认为是有效的。也就是说，如果你已经在你的产品中使用一个密码体制很多年了，而其他某个人声明的专利覆盖了你的这一部分，你将不得不证明你所使用的是先知部分。当然，它的代价是很大的。要请专利律师，而且专利成为了进行合法勒索的便利工具：买一个便宜的授权，或者花十倍的价钱请律师。

24.2 后续

专利系统有一个很古怪的特性。我们并不理解它是怎样来工作的，但它是专利律师惯用的小把戏。对于任何已经申请专利的技术来说，你总要保持一两个专利是“开放”的，也就是说让它保持在申请专利的过程中。当你把专利文件归档时，你要写一个说明文件来阐述技术问题。这个说明文件以后不能改动，但专利声明可以改动。通过过程化的方式很容易来延迟专利的发行过程，因此你可以延迟几个后续专利（基本专利的扩展）的发行。现在假定竞争对手开始使用一种完全不同于你已持有专利的技术，但是与它是密切相关的，那么你应尽力去重写后续专利的声明，使它能覆盖竞争对手的技术。使声明被支持是一个你和专利办公室之间协商的

过程，而且专利办公室并不知道你的竞争对手正在使用什么技术，因此你有很好的机会得到他们的支持，从而就可以让后续专利发行，并且带有直接覆盖竞争对手技术的声明，然后起诉竞争对手专利侵权。这听起来像是欺骗我们，但它确实在应用中出现了，而且显然是合法的。

24.3 含糊性

专利被认为是清晰地阐述了技术，以至于使其他人都能应用专利技术。那是专利交易的一部分。作为对专利技术的反馈，专利持有者对技术有20年的垄断。在实际应用中，专利故意写得很含糊，不清晰，很难读懂。通常，专利持有者不会阐述使用其技术的最好办法，虽然这是专利过程的一个要求。

假如你正在一个系统上工作，你认为一个特定的专利也许有问题。你会拿到专利，读它，并且看看它是否覆盖了你做的工作，对吗？错了！判定一个专利是否覆盖了特定的系统是相当复杂的，最终要由没有技术背景的法官来裁决。预测最终的结果几乎就像由抛硬币决定结果一样。

24.4 读专利

送你一句话作为建议：永远也不要读专利。你可能认为通过读专利来看看它们都覆盖了什么是个好主意，实际上并不是这样的。如果你在不知道的情况下对专利进行了侵权，也许最终要向专利持有者赔偿损失。但是如果他们能证实你是故意侵权（因为你知道他们的专利），你也许要赔偿三倍的损失。因此如果你读了专利，就自动地把侵权的责任增加到了三倍。

现在作为一种真实的讽刺是：即便你读了专利，作为这个领域的专家，你认为你的工作没有被专利所覆盖，法官也许仍然会判定你是故意侵权。你要明白，作为一个专家，你并没有权力来判断专利覆盖了什么，而是专利律师才能判定。因此，如果你想要避免赔偿三倍损失的可能性，就不得不雇佣一个律师来判定是否侵犯了专利权。我们有上百万个专利，雇佣一个律师来读遍所有的专利，这是你不可能负担得起的。

因此，最安全的办法就是从来不去读专利。这样至少你可以声明，你并不是故意去侵犯专利权的。

24.5 授权

如果Alice有一个专利，Bob想要使用那个技术，Alice可以把该专利授权给Bob。这是一件很普通的事情。很多大公司互相给他们的专利授权。他们基本上是对公司所有的专利授权，从而在互相使用专利时不用付钱。小的公司就不得不去申请授权。在很多工程领域，这不是问题。如果你需要使用专利，就能够以合理的方式得到授权。

然而，对于密码学来说情况就不同了。在20世纪80年代，这个领域被一些很基础的关于RSA和DH（那个年代仅有的两个有用的公钥系统）的初始专利所歪曲了。很多年来，对这些

很基础的专利进行授权是相当困难的，而且代价相当高。有很多故事讲的就是某公司尽力去申请授权而最终遭到拒绝的事情。

专利持有者想通过限制授权来控制这些专利控制技术的发展，这就使得其他与之相关的领域得不到这些技术。理论上说，那正是假定专利所支持的事情，但是如果所需要的安全技术因为专利而不能使用，专利持有者拒绝授权，那么整体效果很不好。

最近我们在考虑使用一些新的分组密码模式XCBC、IACBC和OCB时就遇到了困难。有三个发明者，其中至少有两个人的工作是建立在以前工作的基础上的。所有三个领域都有专利，但由于专利发行过程的延迟，还没有专利被发行。因为他们还没有发行，专利声明还鲜为人知，所以要想知道哪个专利覆盖了哪些技术是不可能的。由于在知道其他发明者做什么后，所有的申请者仍然能修改他们的声明（专利声明还没有最后确定），因此很可能使用任何一个专利的任何人都必须要得到所有三个专利的授权。也许仅仅用一个专利的授权就可能使用XCBC，但那需要对专利系统的净化有极大的信心——这并不是我们所建议的。

因此如果你想要使用OCB，也许需要三个专利的授权。那就意味着需要对三个合同协商，而且要有三个不同的律师，假定他们都愿意给专利授权。如果协商用了半年或更多的时间，你也不要感到吃惊。我们已经看到过这样的情况，当漫长的协商过后，专利持有者在最后一刻决定不给专利授权了。这就意味着公司在开发产品过程中遇到了灾难。试图给专利授权，延迟了产品的开发，带来了巨大花费，从而把整个产品线放在了风险中，这种风险是由专利持有者和他们的律师产生的任意行为所带来的。而且，你与他们协商的时间越长，你的处境就会越糟，因为你已经在过程和产品中做了投资，你真的需要让产品立刻投入市场。

让三个发明者凑在一起并把他们的专利混在一起，使得每个人对所有的专利只有一个授权，这似乎是很合逻辑的。有一些人试图去这样组织，但最终我们听到的是发明者不同意这样做。

围绕XCBC、IACBC和OCB的授权问题确实是开发CCM（见8.5节）的动机。CCM的速度仅有它们的一半，但却是无专利的。这种优势远远胜于它所需要的额外计算。

即便是在专利已经发行、它覆盖的技术都很清楚的情况下，对专利进行授权的花费也是很高的。大多数专利持有者使他人得到授权变得很困难。标准的申请授权条款通常是不存在的，或者是太烦琐以至于不能使用。律师一旦开始协商相应的申请授权条款，后面的花费很快就会暴长。专利持有者通常似乎并不愿意让人们用其他方式来使用他们的专利，也不愿意让授权申请容易一些且花费便宜一些。

24.6 专利保护

很多公司申请专利并不是为了应用这些技术，而是为了保护他们自己。如果你有一整套专利，你的竞争对手起诉你侵犯了他的一个专利，那你总能找到一个专利也被他侵犯了，或者至少能断言有一个专利被他侵犯了。这有点像MAD：相互感到放心的破坏。如果你起诉我，我

就会起诉你，我们都会失去大量的金钱（合法的费用和丢失的机会）。有时我们能看到两个公司就卷入了这种小鸡打架的游戏。一旦冷静下来后就会认识到相互的残杀并不能带来任何价值，他们的纠纷才会得到解决，这通常需要一年左右的时间。

24.7 完善专利系统

我们目前的专利系统完全脱离了控制。往好的方面说，专利是一个必不可少的产物。往坏的方面说，它们是欺骗和勒索的合法形式。专利被认为是通过对发明者的研究成果给予奖励来帮助他们。我们认为，当前IT行业专利系统的花费远远超过了它所具有的优势。专利是20年有效的——在IT世界中相当于永恒。得到一个专利要用几年的时间，这能转换成该行业中的几个产品循环。IT工业的时间跨度是如此之快，以至于专利系统不能跟上它。

当然，有一群人不断地从专利系统中受益：那就是律师。

一个经常听到的说法是，专利可以保护小的发明者来抵抗大公司。这是欺骗。你也许能找到几个这样的例子，但大多数时间它却是以另外一种方式来工作的。一个发明者或者一个小公司开发了一个真正的新产品，他们投入了大量时间和金钱并把它投入了市场。但是一旦产品开始威胁到大公司的市场，他们就会被专利的法律起诉所威胁。小公司根本无法负担法庭官司的费用，这就使得他们被敲诈勒索。我们知道至少有一个这样的小公司经历了这种事情，最终他们不得不把产品以很低的价格卖给了大公司。目前的专利系统给了大公司巨大的能量，他们有雇来的专利律师，而且有足够多的金钱可以对其他小公司施以专利起诉的威胁。

我们认为IT行业如果没有专利，将会更好一些。我们有些朋友有不同的意见，当然这是一个开放式的争论。但是当前的系统确实不能正常工作。

最小限度地说，我们想要提高专利系统的工作效率。发行专利在三个月之内完成，而且把专利的有效期限制在三年到五年。允许专利面临任何人基于先知部分的挑战，即使是在专利发行之后。创建一个系统来决定某个特定的产品是否侵犯了专利权，系统的工作时间跨度是几个月而不是几年。在IT行业，如果有判定制度说某人侵犯了五年前的专利，这种制度是没有好处的。你需要在行业的时间跨度范围内给出答案，那至多是几个月。

当然，这仅仅是我们的观点。专利系统还没有完善，因为在这个领域没有政治上的利益，因此我们必须要依赖这个系统，那可能就意味着人们总是要对发明的技术申请专利，而且只是为了保护自己抵抗竞争对手。密码学的设计者也要用一些时间去考虑如何避免涉及专利问题。

24.8 不予承诺

我们不是律师，甚至看起来也不像律师，因此不要把我们的建议当成是合法的。换句话说，你要自己拿主意，除非你以每小时几百美元的费用雇用了专利律师。

第25章　专　　家

密码学界有一些很奇怪的事情：每个人都认为自己知道足够多的知识，可以设计和构建他们的系统。我们从来不会让一个二年级的物理系学生去设计原子能动力工厂，也不会让一个声称自己找到了心脏手术新办法的正在接受培训的护士去给病人做手术。然而，读了一两本书的人却认为他们能设计自己的密码系统。更糟糕的是，他们有时能让管理层、风险投资者、甚至是客户相信他们的设计是最好的一个。

在密码学家中，Bruce的第一本书《应用密码学》（"Applied Cryptography"）[87，88]，既是著名的又是声名狼籍的。它因为使几万人开始注意密码学而著名，同时又由于这些人设计和应用的系统而声名狼籍。

一个最近的例子是802.11——无线网络标准。初始的设计包含一个安全的信道，叫做有线等价的私有性（WEP, wired equivalent privacy），它用来加密和认证无线通信。标准是由一个不包含任何专家的委员会设计的。结果很可怕。使用RC4加密算法的决定不是最好的一个，但本身并没有致命的缺陷。然而，RC4是流密码，需要惟一的瞬时值。WEP没有给瞬时值分配足够的比特，结果是相同的瞬时值不得不被重新使用，这就导致了很多包是用相同的密钥流加密的。它使RC4流密码的加密特性变为无效，聪明的攻击者会攻破这种加密。一个更微妙的缺陷是，在使用密钥和瞬时值作为RC4密钥之前，没有对它们取Hash值，这最终会导致密钥恢复的攻击[36]。CRC计数器被用于认证，但由于CRC计算是线性的，很容易不经过检测就修改任何数据包（使用线性算子）。一个共享密钥被网络中所有的用户使用，这使得密钥更新很难完成。网络密码口令直接被用做通信中的加密密钥，而不使用任何的密钥协商协议。最后，加密在默认状态下是关闭的，这意味着大多数应用从开始就都不使用加密。WEP不仅仅是被攻破了，而且是被完全地攻破了。

设计一个WEP的替代物并不容易，因为它必须要与现有的硬件配套。但是我们没有选择，因为原始标准的安全性简直太差了。它的替代物叫做WPA，现在应该是可用的。

WEP的故事不是一个例外。因为802.11是一个成功的产品，所以更容易被采用，我们在其他系统中也看到过很多类似的情况。一个同事曾经告诉Bruce："这个世界充满了不安全系统，这些系统是由读《应用密码学》一书的人设计的。"

本书有可能也会产生同样的影响。

这会使本书成为一本很危险的书。很多人将阅读这本书，然后去设计密码算法和协议。当他们完成设计时，会有一些东西看起来很好，也许甚至能够工作，但其安全性却会带来灾难。

他们的设计也许能有70%是正确的，如果幸运的话，也许能达到90%。但密码学中不可能有完全正确的事。一个安全的系统只是同它的最弱连接一样的强壮。为了做到安全，一切都必须是正确的。而所有这些是不能仅从书本中学到的。

既然本书可能会导致坏的系统，为什么我们还要写呢？我们这本书是针对那些想知道怎样设计密码系统的人而写的，我们不知道还有哪本书比本书更合适。应该把这本书看成是这个领域的一个介绍，虽然它并不是一本手册或教科书。我们这本书也是写给参加项目研发的工程师写的。安全系统的每个部分都是至关重要的，工作在某个项目中的每个人对它所包含的安全观点和技术都应该有一个基本的理解。这包括程序员、测试员、技术起草者、管理员甚至是销售人员。每个人都需要了解足够的安全知识来做好自己的工作。我们希望本书能提供足够的密码学实践的背景。

如果我们能给你一点儿建议的话，就是要尽可能地聘用密码学专家。如果你的项目包含了密码学，那么就需要一个有经验的密码设计者。在项目的开始就需要一个这样的专家。你同密码学专家商量得越早，项目的最终花费就越低，而且项目进展就越顺利。很多次项目进展得已经很顺利了，但却要对在项目设计或应用阶段留下的漏洞进行补救。最终结果总是代价很大，或者更改了项目时间表和花费，或者没有满足产品的安全性。

要想做好密码系统是相当困难的，即使是由专家设计的系统也经常失败。这与你是多么聪明或者在其他领域有多少经验都没有关系。设计和应用密码系统要求专业的知识和经验，得到经验的惟一办法就是一遍又一遍地去做。当然，专家也会犯一些错误，那么为什么还要用他呢？这与你会选择一个好的外科大夫来给你做手术是一个道理。不是他们不出错，而是他们犯的错误少，而且危害小。他们会以保守的方式来工作，小错误不会导致灾难性的后果，而且他们还知道怎样去面对失败。

应用密码系统几乎像设计密码系统一样的专业。密码系统的设计者是能够雇到的，而密码系统的应用者却难以雇到，部分原因是需求量太大。一个设计者设计的系统需要由10个到20个应用者去应用。大多数人并不想把密码系统的应用作为一个专业来对待。程序员会从数据库编程转向GUI的工作，然后转向密码学应用。诚然，数据库编程和GUI工作也是很专业的，但一个有经验的程序员经过一些学习就能很好地工作，然而对密码学应用来说就不是这样了，系统的每个组成部分都必须是正确的。

我们所知道的应用密码系统的最好办法是采用有竞争力的程序员，为了完成某个任务而对他们进行特殊培训。这本书可以是培训的一部分，但主要还是需要经验和正确的思想意识。就像其他的专业IT技能一样，要想真正掌握，需要几年的时间。由于要花很长的时间来积累经验，你必须在他们成熟之前一直把他们留在公司。那是另一个问题，我们很高兴留给其他人去解决。

也许比本书更重要的或者比任何书都重要的是项目文化。“安全第一”不应该仅仅是一个口号，它应该成为项目和项目团队的基本共识。每个人在生活、呼吸、谈话和思考时想的都应

该是安全。这是很难达到的，但确实能够达到。DigiCash在20世纪90年代就有一个这样的团队，航空工业就有一个类似的很普遍的安全文化，这在短时期内是不可能达到的，但通过不断努力就肯定能做到。对于团队中的多数技术人员来说，本书介绍的仅仅是关于最重要的安全观点的基础。

正如Bruce在《秘密与谎言》(“Secrets and Lies”)一书中所说：“安全是一个过程，而不是一个产品。”除了安全文化，我们也需要一个安全的过程。航空工业就有一个扩展的安全过程。大多数IT企业甚至都没有生产软件的过程，更不用说生产高质量软件的过程了，生产安全软件的过程就更少得可怜了。写一个好的安全软件可能是在IT行业现有状态的能力之外，但这并不意味着我们就应该放弃。由于信息技术对于我们的基础设施、我们的自由、我们的安全都是至关重要的，因此必须要改进系统的安全性，我们必须要尽最大的努力去做。

我们希望本书通过给那些从事安全系统方面工作的人传授密码学实践的基础知识，能够对安全系统的改善做出一些贡献。

致　谢

本书是基于作者在密码学方面多年的工作经验写出的。非常感谢和我们一起工作的所有人，他们给我们的工作带来了很多乐趣，并帮助我们完成了本书。我们还想感谢我们的客户，是他们为我们能继续密码学的研究提供了资金，而且提供了撰写本书所必需的实际经验。

此外，在本书成书过程中，Beth Friedman做了非常宝贵的编辑工作，Denise Dick的校正工作改进了我们的文稿，John Kelsey对密码学的内容提供了有价值的反馈。因特网使我们的合作成为可能。我们还要感谢Carol Long和Wiley团队的其他人，他们使我们的想法得以实现。

最后，我们要感谢世界上的所有程序员，他们一直在努力地编写密码学的代码，并对其他人免费公开。

PDG

参考文献

[1] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems.* John Wiley & Sons, Inc., 2001. ISBN 0-471-38922-6.

[2] Ross Anderson, Eli Biham, and Lars Knudsen. Serpent: A proposal for the Advanced Encryption Standard. In National Institute of Standards and Technology [72]. See `http://www.cl.cam.ac.uk/~rja14/serpent.html` or `http://www.nist.gov/aes`.

[3] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying Hash Functions for Message Authentication. In Neal Koblitz, editor, *Advances in Cryptology—CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 1996.

[4] Mihir Bellare, Joe Kilian, and Phillip Rogaway. The Security of Cipher Block Chaining. In Yvo G. Desmedt, editor, *Advances in Cryptology—CRYPTO '94*, volume 839 of *Lecture Notes in Computer Science*, pages 341–358. Springer-Verlag, 1994.

[5] Charles H. Bennett and Gilles Brassard. An update on quantum cryptography. In G.R. Blakley and David Chaum, editors, *Advances in Cryptology, Proceedings of CRYPTO 84*, volume 196 of *Lecture Notes in Computer Science*, pages 475–480. Springer-Verlag, 1984.

[6] Eli Biham, Orr Dunkelman, and Nathan Keller. The Rectangle Attack—Rectangling the Serpent. In Birgit Pfitzmann, editor, *Advances in Cryptology—EUROCRYPT 2001*, volume 2045 of *Lecture Notes in Computer Science*, pages 340–357. Springer-Verlag, 2001.

[7] Eli Biham. New Types of Cryptanalytic Attacks Using Related Keys. In Tor Helleseth, editor, *Advances in Cryptology—EUROCRYPT '93*, volume 765 of *Lecture Notes in Computer Science*, pages 398–409. Springer-Verlag, 1993.

[8] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: Fast and Secure Message Authentication. In Michael Wiener, editor, *Advances in Cryptology—CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 216–233. Springer-Verlag, 1999.

[9] Jurjen Bos. Booting problems with the JEC computer. Personal communications, 1983.

[10] Jurjen Bos. *Practical Privacy.* PhD thesis, Eindhoven University of Technology, 1992. Avalable from `http://www.macfergus.com/niels/lib/bosphd.html`.

[11] Gilles Brassard and Claude Crépeau. Quantum Bit Commitment and Coin Tossing Protocols. In A.J. Menezes and S.A. Vanstone, editors, *Advances in Cryptology—CRYPTO '90*, volume 537 of *Lecture Notes in Computer Science*, pages 49–61. Springer-Verlag, 1990.

[12] Karl Brincat and Chris J. Mitchell. New CBC-MAC forgery attacks. In V. Varadharajan and Y. Mu, editors, *Information Security and Privacy, ACISP 2001*, volume 2119 of *Lecture Notes in Computer Science*, pages 3–14. Springer-Verlag, 2001.

[13] Carolynn Burwick, Don Coppersmith, Edward D'Avignon, Rosario Gennaro, Shai Halevi, Charanjit Jutla, Stephen M. Matyas Jr., Luke O'Connor, Mohammad Peyravian, David Safford, and Nevenko Zunic. MARS—a candidate cipher for AES. In National Institute of Standards and Technology [72]. See `http://www.research.ibm.com/security/mars.html` or `http://www.nist.gov/aes`.

[14] Christian Cachin. *Entropy Measures and Unconditional Security in Cryptography.* PhD thesis, ETH, Swiss Federal Institute of Technology, Zürich, 1997. See `ftp://ftp.inf.ethz.ch/pub/publications/dissertations/th12187.ps.gz`.

[15] Lewis Carroll. *The Hunting of the Snark: an Agony, in Eight Fits.* Macmillan and Co., London, 1876.

[16] Florent Chabaud and Antoine Joux. Differential Collisions in SHA-0. In Hugo Krawczyk, editor, *Advances in Cryptology—CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 56–71. Springer-Verlag, 1998.

[17] Nicolas Courtois and Josef Pieprzyk. Cryptanalysis of Block Ciphers with Overdefined Systems of Equations. Cryptology ePrint Archive, Report 2002/044, 2002. `http://eprint.iacr.org/`.

[18] Joan Daemen and Vincent Rijmen. AES Proposal: Rijndael. In National Institute of Standards and Technology [72]. See `http://www.esat.kuleuven.ac.be/~rijmen/rijndael/` or `http://www.nist.gov/aes`.

[19] Don Davis, Ross Ihaka, and Philip Fenstermacher. Cryptographic Randomness from Air Turbulence in Disk Drives. In Yvo G. Desmedt, editor, *Advances in Cryptology—CRYPTO '94*, volume 839 of *Lecture Notes in Computer Science*, pages 114–120. Springer-Verlag, 1994.

[20] Bert den Boer and Antoon Bosselaers. Collisions for the compression function of MD5. In Tor Helleseth, editor, *Advances in Cryptology—EUROCRYPT '93*, volume 765 of *Lecture Notes in Computer Science*, pages 293–304. Springer-Verlag, 1993.

[21] Whitfield Diffie and Martin E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976.

[22] Whitfield Diffie and Martin E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976.

[23] Edsger W. Dijkstra. The Humble Programmer. *Communications of the ACM*, 15(10):859–866, 1972. Also published as EWD340, `http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD340.PDF`.

[24] Giovanni Di Crescenzo, Niels Ferguson, Russel Impagliazzo, and Markus Jakobsson. How To Forget a Secret. In Christoph Meinel and Sophie Tison, editors, *STACS 99*, volume 1563 of *Lecture Notes in Computer Science*, pages 500–509. Springer-Verlag, 1999.

[25] Hans Dobbertin. Cryptanalysis of MD4. *J. Cryptology*, 11(4):253–271, 1998.

[26] Stephen R. Dussé and Burton S. Kaliski Jr. A Cryptographic Library for the Motorola DSP56000. In I.B. Damgård, editor, *Advances in Cryptology—EUROCRYPT '90*, volume 473 of *Lecture Notes in Computer Science*, pages 230–244. Springer-Verlag, 1990.

[27] Morris Dworkin. *Recommendation for Block Cipher Modes of Operation—Methods and Techniques*. National Institute of Standards and Technology, December 2001. Available from `http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf`.

[28] Carl Ellison. Improvements on Conventional PKI Wisdom. In Sean Smith, editor, *1st Annual PKI Research Workshop—Proceedings*, pages 165–175, 2002. Available from `http://www.cs.dartmouth.edu/~pki02/Ellison/`.

[29] Jan-Hendrik Evertse and Eugène van Heyst. Which New RSA-Signatures Can Be Computed from Certain Given RSA-Signatures? *J. Cryptology*, 5(1):41–52, 1992.

[30] H. Feistel, W.A. Notz, and J.L. Smith. Some Cryptographic Techniques for Machine-to-Machine Data Communications. *Proceedings of the IEEE*, 63(11):1545–1554, 1975.

[31] Niels Ferguson, John Kelsey, Stefan Lucks, Bruce Schneier, Mike Stay, David Wagner, and Doug Whiting. Improved Cryptanalysis of Rijndael. In Bruce Schneier, editor, *Fast Software Encryption, 7th International Workshop, FSE*

2000, volume 1978 of *Lecture Notes in Computer Science*, pages 213–230. Springer-Verlag, 2000.

[32] Niels Ferguson, John Kelsey, Bruce Schneier, and Doug Whiting. A Twofish Retreat: Related-Key Attacks Against Reduced-Round Twofish. Twofish Technical Report 6, Counterpane Systems, February 2000. See `http://www.counterpane.com/twofish.html`.

[33] Niels Ferguson and Bruce Schneier. A Cryptographic Evaluation of IPsec, 1999. See `http://www.counterpane.com/ipsec.html`.

[34] Niels Ferguson, Richard Schroeppel, and Doug Whiting. A simple algebraic representation of Rijndael. In Serge Vaudenay and Amr M. Youssef, editors, *Selected Areas in Cryptography, 8th Annual International Workshop, SAC 2001*, volume 2259 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[35] Niels Ferguson. Collision attacks on OCB. Unpublished manuscript, 2002.

[36] Scott Fluhrer, Itsik Mantin, and Adi Shamir. Weaknesses in the Key Schedule Algorithm of RC4. In Serge Vaudenay and Amr M. Youssef, editors, *Selected Areas in Cryptography, 8th Annual International Workshop, SAC 2001*, volume 2259 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[37] Alan O. Freier, Philip Karlton, and Paul C. Kocher. The SSL Protocol, Version 3.0. Internet draft, Transport Layer Security Working Group, November 18, 1996. Available from `http://home.netscape.com/eng/ssl3/`.

[38] Ian Goldberg and David Wagner. Randomness and the Netscape Browser. *Dr. Dobb's Journal*, pages 66–70, January 1996. Available from `www.cs.berkeley.edu/~daw/papers/ddj-netscape.html`.

[39] Peter Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. In *USENIX Security Symposium Proceedings*, 1996. Available from `http://www.auckland.ac.nz/~pgut001`.

[40] Peter Gutmann. X.509 Style Guide, October 2000. Available from `http://www.cs.auckland.ac.nz/~pgut001/pubs/x509guide.txt`.

[41] D. Harkins and D. Carrel. The Internet Key Exchange (IKE). RFC 2409, November 1998.

[42] Intel. *Intel 82802 Firmware Hub: Random Number Generator, Programmers Reference Manual*, December 1999. Available from the Intel web site.

[43] International Telecommunication Union. *X.680-X.683: Abstract Syntax Notation One (ASN.1), X.690-X.693: ASN.1 encoding rules*, 2002. Available from `www.itu.int/ITU-T/studygroups/com17/languages/x680-x693_0702.pdf`.

[44] Jakob Jonsson. On the Security of CTR+CBC-MAC. In *Selected Areas in Cryptography, 9th Annual International Workshop, SAC 2002*, 2002. See `http://csrc.nist.gov/encryption/modes/proposedmodes/ccm/ccm-ad.pdf`.

[45] Robert R. Jueneman. Analysis of Certain Aspects of Output Feedback Mode. In David Chaum, Ronald L. Rivest, and Alan T. Sherman, editors, *Advances in Cryptology, Proceedings of Crypto 82*, pages 99–128. Plenum Press, 1982.

[46] David Kahn. *The Codebreakers, The Story of Secret Writing*. Macmillan Publishing Co., New York, 1967.

[47] John Kelsey, Bruce Schneier, and Niels Ferguson. Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator. In Howard Heys and Carlisle Adams, editors, *Selected Areas in Cryptography, 6th Annual International Workshop, SAC '99*, volume 1758 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[48] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Cryptanalytic Attacks on Pseudorandom Number Generators. In Serge Vaudenay, editor, *Fast Software Encryption, 5th International Workshop, FSE'98*, volume 1372 of *Lecture Notes in Computer Science*, pages 168–188. Springer-Verlag, 1998.

[49] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Side Channel Cryptanalysis of Product Ciphers. *Journal of Computer Security*, 8(2–3):141–158, 2000. See also `http://www.counterpane.com/side_channel.html`.

[50] John Kelsey, Bruce Schneier, and David Wagner. Key-Schedule Cryptanalysis of IDEA, G-DES, GOST, SAFER, and Triple-DES. In Neal Koblitz, editor, *Advances in Cryptology—CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 237–251. Springer-Verlag, 1996.

[51] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. RFC 2401, November 1998.

[52] John Keogh. *Circular transportation facilitation device*. Australian Patent Office, August 2001. Granted Innovation Patent No. AU2001100012 A4, available from `http://www.ipmenu.com/archive/AUI_2001100012.pdf`.

[53] Joe Killian and Phillip Rogaway. How to Protect DES Against Exhaustive Key Search. In Neal Koblitz, editor, *Advances in Cryptology—CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 252–267. Springer-Verlag, 1996.

[54] Lars R. Knudsen and Vincent Rijmen. Two Rights Sometimes Make a Wrong. In *Workshop on Selected Areas in Cryptography (SAC '97)*, pages 213–223, 1997. See `http://adonis.ee.queensu.ca:8000/sac/sac97/papers.html`.

[55] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, 1981.

[56] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Neal Koblitz, editor, *Advances in Cryptology—CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer-Verlag, 1996.

[57] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In Michael Wiener, editor, *Advances in Cryptology—CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer-Verlag, 1999.

[58] J. Kohl and C. Neuman. The Kerberos Network Authentication Service (V5). RFC 1510, September 1993.

[59] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, February 1997.

[60] T. Krovetz, J. Black, S. Halevi, A. Hevia, H. Krawczyk, and P. Rogaway. UMAC: Message Authentication Code using Universal Hashing. RFC draft draft-krovetz-umac-01.txt, 2000.

[61] Xuejia Lai and James L. Massey. A Proposal for a New Block Encryption Standard. In I.B. Damgård, editor, *Advances in Cryptology—EUROCRYPT '90*, volume 473 of *Lecture Notes in Computer Science*, pages 389–404. Springer-Verlag, 1990.

[62] Xuejia Lai, James L. Massey, and Sean Murphy. Markov Ciphers and Differential Cryptanalysis. In D.W. Davies, editor, *Advances in Cryptology—EUROCRYPT '91*, volume 547 of *Lecture Notes in Computer Science*, pages 17–38. Springer-Verlag, 1991.

[63] Arjen K. Lenstra and Eric R. Verheul. Selecting Cryptographic Key Sizes. *J. Cryptology*, 14(4):255–293, August 2001.

[64] T. Matsumoto, H. Matsumoto, K. Yamada, and S. Hoshino. Impact of Artificial "Gummy" Fingers on Fingerprint Systems. In *Proceedings of SPIE, Vol #4677, Optical Security and Counterfeit Deterrence Techniques IV*, to appear. See also `www.itu.int/itudoc/itu-t/workshop/security/present/s5p4.pdf`.

[65] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996. ISBN 0-8493-8523-7.

[66] David L. Mills. Network Time Protocol (Version 3). RFC 1305, March 1992.

[67] D. Mills. Simple Network Time Protocol (SNTP) Version 4. RFC 2030, October 1996.

[68] P. Montgomery. Modular Multiplication without Trial Division. *Mathematics of Computation*, 44(170):519–521, 1985.

[69] National Institute of Standards and Technology. *DES Modes of Operation*, December 2, 1980. FIPS PUB 81, available from `http://www.itl.nist.gov/fipspubs/`.

[70] National Institute of Standards and Technology. *Data Encryption Standard (DES)*, December 30, 1993. FIPS PUB 46-2, available from `http://www.itl.nist.gov/fipspubs/`.

[71] National Institute of Standards and Technology. *Secure Hash Standard*, April 17, 1995. FIPS PUB 180-1, available from `http://www.itl.nist.gov/fipspubs/`.

[72] National Institute of Standards and Technology. *AES Round 1 Technical Evaluation, CD-1: Documentation*, August 1998. See `http://www.nist.gov/aes`.

[73] National Institute of Standards and Technology. *Data Encryption Standard (DES)*, 1999. DRAFT FIPS PUB 46-3, available from `http://csrc.ncsl.nist.gov/fips/`.

[74] National Institute of Standards and Technology. *Proc. 3rd AES candidate conference*, April 2000.

[75] National Institute of Standards and Technology. *Secure Hash Standard (draft)*, 2001. Draft FIPS PUB 180-2, available from `http://csrc.nist.gov/encryption/shs/dfips-180-2.pdf`.

[76] Roger M. Needham and Michael D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[77] Bart Preneel and Paul C. van Oorschot. On the Security of Two MAC Algorithms. In Ueli Maurer, editor, *Advances in Cryptology—EUROCRYPT '96*, volume 1070 of *Lecture Notes in Computer Science*, pages 19–32. Springer-Verlag, 1996.

[78] Ronald L. Rivest, M.J.B. Robshaw, R. Sidney, and Y.L. Yin. The RC6 Block Cipher. In National Institute of Standards and Technology [72]. See `http://www.rsasecurity.com/rsalabs/rc6/` or `http://www.nist.gov/aes`.

[79] Ronald L. Rivest. The MD4 Message Digest Algorithm. In A.J. Menezes and S.A. Vanstone, editors, *Advances in Cryptology—CRYPTO '90*, volume 537 of *Lecture Notes in Computer Science*, pages 303–311. Springer-Verlag, 1990.

[80] Ronald L. Rivest. The RC5 Encryption Algorithm. In B. Preneel, editor, *Fast Software Encryption, Second International Workshop, FSE'94*, volume 1008 of *Lecture Notes in Computer Science*, pages 86–96. Springer-Verlag, 1995.

[81] Ronald Rivest, Adi Shamir, and Leonard Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21:120–126, February 1978.

[82] R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321, April 1992.

[83] Phillip Rogaway, Mihir Bellare, John Black, and Ted Krovetz. OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption. In *Eighth ACM Conference on Computer and Communications Security (CCS-8)*, pages 196–205. ACM, ACM Press, 2001.

[84] Phillip Rogaway, Mihir Bellare, John Black, and Ted Krovetz. OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption, September 2001. Available from `http://www.cs.ucdavis.edu/~rogaway`.

[85] RSA Laboratories. *PKCS #1 v2.1: RSA Cryptography Standard*, January 2001. Available from `http://www.rsasecurity.com/rsalabs/pkcs`.

[86] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. *The Twofish Encryption Algorithm, A 128-Bit Block Cipher*. Wiley, 1999.

[87] Bruce Schneier. *Applied Cryptography, Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., 1994. ISBN 0-471-59756-2.

[88] Bruce Schneier. *Applied Cryptography, Second Edition, Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., 1996. ISBN 0-471-12845-7.

[89] Bruce Schneier. *Secrets and Lies, Digital Security in a Networked World*. John Wiley & Sons, Inc., 2000. ISBN 0-471-25311-1.

[90] Dr. Seuss. *Horton Hears a Who!* Random House, 1954.

[91] C.E. Shannon. A Mathematical Theory of Communication. *The Bell Systems Technical Journal*, 27:370–423 and 623–656, July and October 1948. See `http://cm.bell-labs.com/cm/ms/what/shannonday/paper.html`.

[92] David Wagner, Niels Ferguson, and Bruce Schneier. Cryptanalysis of FROG. In *Proc. 2nd AES candidate conference*, pages 175–181. National Institute of Standards and Technology, March 1999.

[93] David Wagner and Bruce Schneier. Analysis of the SSL 3.0 protocol. In *Proceedings of the Second USENIX Workshop on Electronic Commerce*, pages 29–40. USENIX Press, November 1996. Revised version available from `http://www.counterpane.com`.

[94] Doug Whiting, Russ Housley, and Niels Ferguson. Counter with CBC-MAC (CCM), June 2002. See `http://csrc.nist.gov/encryption/modes/proposedmodes/ccm/ccm.pdf`.

[95] Michael J. Wiener. Cryptanalysis of short RSA secret exponents. *IEEE Transactions on Information Theory*, 36(3):553–558, May 1990.

[96] Robert S. Winternitz. Producing a One-way Hash Function from DES. In David Chaum, editor, *Advances in Cryptology, Proceedings of Crypto 83*, pages 203–207. Plenum Press, 1983.

[97] Thomas Wu. The Secure Remote Password Protocol. In *Proceedings of the 1998 Network and Distributed System Security (NDSS'98) Symposium*, March 1998.

[General Information]
书名=
作者=(美)Niels Ferguson，(美) Bruce Schneier著；张振峰，徐静，李红达等译
页数=244
出版社=北京市：电子工业出版社
出版日期=2005.07
SS号=
DX号=000005752879
URL=http://book.szdnet.org.cn/bookDetail.jsp?dxNumber=000005752879&d=B69DB8647BD932206E670D469727BD8A